African Virtual University

Applied Computer Science: CSI 3104

OBJECT ORIENTED ANALYSIS AND DESIGN

Dr Ellen Ambakisye Kalinga

Foreword

The African Virtual University (AVU) is proud to participate in increasing access to education in African countries through the production of quality learning materials. We are also proud to contribute to global knowledge as our Open Educational Resources are mostly accessed from outside the African continent.

This module was developed as part of a diploma and degree program in Applied Computer Science, in collaboration with 18 African partner institutions from 16 countries. A total of 156 modules were developed or translated to ensure availability in English, French and Portuguese. These modules have also been made available as open education resources (OER) on oer.avu.org.

On behalf of the African Virtual University and our patron, our partner institutions, the African Development Bank, I invite you to use this module in your institution, for your own education, to share it as widely as possible and to participate actively in the AVU communities of practice of your interest. We are committed to be on the frontline of developing and sharing Open Educational Resources.

The African Virtual University (AVU) is a Pan African Intergovernmental Organization established by charter with the mandate of significantly increasing access to quality higher education and training through the innovative use of information communication technologies. A Charter, establishing the AVU as an Intergovernmental Organization, has been signed so far by nineteen (19) African Governments - Kenya, Senegal, Mauritania, Mali, Cote d'Ivoire, Tanzania, Mozambique, Democratic Republic of Congo, Benin, Ghana, Republic of Guinea, Burkina Faso, Niger, South Sudan, Sudan, The Gambia, Guinea-Bissau, Ethiopia and Cape Verde.

The following institutions participated in the Applied Computer Science Program: (1) Université d'Abomey Calavi in Benin; (2) Université de Ougagadougou in Burkina Faso; (3) Université Lumière de Bujumbura in Burundi; (4) Université de Douala in Cameroon; (5) Université de Nouakchott in Mauritania; (6) Université Gaston Berger in Senegal; (7) Université des Sciences, des Techniques et Technologies de Bamako in Mali (8) Ghana Institute of Management and Public Administration; (9) Kwame Nkrumah University of Science and Technology in Ghana; (10) Kenyatta University in Kenya; (11) Egerton University in Kenya; (12) Addis Ababa University in Ethiopia (13) University of Rwanda; (14) University of Dar es Salaam in Tanzania; (15) Universite Abdou Moumouni de Niamey in Niger; (16) Université Cheikh Anta Diop in Senegal; (17) Universidade Pedagógica in Mozambique; and (18) The University of the Gambia in The Gambia.

Bakary Diallo

The Rector

African Virtual University

Production Credits

Author

Dr. Ellen Ambakisye Kalinga

Peer Reviewer

Victor Odumuyiwa

AVU - Academic Coordination

Dr. Marilena Cabral

Overall Coordinator Applied Computer Science Program

Prof Tim Mwololo Waema

Module Coordinator

Jules Degila

Instructional Designers

Elizabeth Mbasu

Diana Tuel

Benta Ochola

Media Team

Sidney McGregor

Barry Savala

Edwin Kiprono

Kelvin Muriithi

Victor Oluoch Otieno

Michal Abigael Koyier

Mercy Tabi Ojwang

Josiah Mutsogu

Kefa Murimi

Gerisson Mulongo

Copyright Notice

Supported By

AVU Multinational Project II funded by the African Development Bank.

Table of Contents

Course Overview

Welcome to Object-oriented Analysis and Design

Object-Oriented Analysis and Design module teaches students on how to effectively use object-oriented technologies and software modeling as applied to software development process with the help of Unified Modeling Language (UML). UML is the standard language for object-oriented analysis and design. UML is used throughout the software development life cycle to capture and communicate analysis and design artifacts. In this course you will use graphical modeling language, to communicate concepts, decisions, understand the problem, propose the solution and manage complexity of artifacts. Finally, the module teaches students to use patterns and frameworks when building reusable software components.

Prerequisites

- HJ08: Object Oriented Programming Module
- PJ16: Software Engineering

Materials

The materials required to complete this course are:

- Text/Reading books
- Lecture notes
- Modelling tools such as Visio studio, ArgoUML, Smart Drawer
- Computer
- Internet Connection

Course Goals

Upon completion of this course the learner should be able to:

i. Design quality blueprints for enterprise-level systems that incorporate architectural and design patterns

ii. Develop the supporting documentation for the requirement, analysis, and design phases of a software development project using an Object-Oriented Analysis and Design (OOAD) method in conjunction with the Unified Modelling Language (UML)

iii. Creating collaboration diagrams and assigning responsibilities to objects with the use of GRASP - Patterns of General Principles in Assigning Responsibilities

iv. Map a design to Code,

v. Demonstrate the analysis and design techniques in the development of enterprise-level software systems.

Units

Unit 0: Basic Knowledge

The unit reviews the basic knowledge in software engineering. The unit starts by reviewing software development life cycles (SDLC) or software development process. Software development methodologies, categorized as structured approach and object-oriented approach are differentiated. The unit emphasizes the importance of learning-by-doing so as to grasp the Object-oriented approach to software development through a number of case projects; two for a class and others for group of students.

Unit 1: Principles of Object Oriented

The unit presents the object oriented principles: Abstraction, Hierarchy, Modularity, and Encapsulation and fundamental concepts of Object Orientation: Objects, Classes, Inheritance and Polymorphism. This unit explains associations and links: Association, Composition, Inheritance or Generalization, Aggregation. It explains Super Class, Sub Class, and Multiplicity.

Unit 2: Fundamentals of UML

This unit gives an overview of the Unified Modeling Language (UML). UML has become the universally-accepted language for software design blueprints. It has emerged as the standard notation for modeling. UML is the visual language used to convey design ideas, which emphasizes how developers really apply frequently used UML elements, in software development process.

Unit 3: Object-Oriented Analysis

This unit explains how requirement specifications can be obtained by involving use cases, identified classes and considering system operations and operation contracts which are obtained from system sequence model.

Unit 4: Object-Oriented Design

After identifying requirements and creating a domain model, then methods are added to the software classes, and messaging between the objects to fulfill the requirements are defined. Deciding what methods belong where, and how the objects should interact, is terribly important. This is the heart of what it means to develop an object-oriented system, apart from drawing domain model diagrams.

Unit 5: Implementing/Translating Object-Oriented Designs into Programming Languages

The UML artifacts created during the design phase: the interaction diagrams and Design Class Diagram (DCDs) will be used as input to the code generation process. Implementation consists of artifacts such as the source code, database definitions, JSP/XML/HTML pages etc. Code created is part of the implementation model. Java object oriented programming language has been used to demonstrate the mapping of the design to code.

Assessment

Formative assessments, used to check learner progress, are included in each unit. This part will include the assessment of the individual/group case project to be submitted at the end of the module

Summative assessments, such as final tests and assignments, are provided at the end of each module and cover knowledge and skills from the entire module.

Summative assessments are administered at the discretion of the institution offering the course. The suggested assessment plan is as follows:

1	Consultation of materials and resources	20 marks
1	Individual/group (Hands-on) case projects	40 marks
3	Formative and summative assessments	40 marks

Unit	Activities	Estimated time
Unit 0	Basic Knowledge:	1 Week (2 Hrs)
	Activity 0.1 – Software Development	
Unit I	Principles of object modelling:	2 Weeks (16 Hrs)
	Activity 1.1 – Case Studies	
	Activity 1.2 – Software development	
	Activity 1.3 –Principles and Concepts of Object Oriented	
	Activity 1.4 – Associations and Links	
Unit II	Fundamentals of UML:	2 Weeks (24 Hrs)
	Activity 2.1 – UML as a Modelling Tool	
	Activity 2.2: UML Diagrams	
Unit III	Object-oriented analysis:	4 Weeks (32 Hrs)
	Activity 3.1 – What are Requirements	
	Activity 3.2 – Use Case	
	Activity 3.3 – Conceptual Modelling	
	Activity 3.4 – System Behaviour: System Sequence Diagrams and Operations	

Unit IV	Object-oriented design:	4 Weeks (24 Hrs)
	Activity 4.1 – Interactive Diagrams – Collaboration Diagram	
	Activity 4.2 – Overview of Design Phase	
	Activity 4.3 – Application of Patterns in Design	
	Activity 4.4 - Design Class Diagram	
Unit V	Implementing/translating object-oriented designs into programming languages	3 Weeks (24 Hrs)
	Activity 5.1 - Notation for Class Interface Details	
	Activity 5.2 - Mapping a Design to Code	

Readings and Other Resources

The readings and other resources in this course are:

Unit 0

Required readings and other resources:

- Bjork R. C., (2004). ATM Simulation. ATM Online, URL: http://www.math-cs.gordon.edu/courses/cps211/ATMExample/
- Liu, Z. (2001). Object-Oriented Software Development Using UML", March, The United University – International Institute for Software Technology (UNU/IIST), Report No. 229.
- Nellen, T. and Mayo, L. (2000), "We Learn by Doing", URL: http://english.ttu.edu/kairos/5.1/coverweb/nellenmayo/doing.html.

Unit 1

Required readings and other resources:

- Ariadne Training (2001), "UML Applied Object Oriented Analysis and Design Using the UML", Ariadne Training Limited

- Liu Z., (2001), "Object-Otiented Software Development Using UML", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.
- Ojo A. and Estevez E., (2005), "Object-Oriented Analysis and Design with UML", Training Course, The United Nations University, UNU-IIST International Institute for Software Technology, e-Macao Report 19, Version 1.0, October.
- Sommerville Ian (2000), "Software Engineering (6th Edition)". Addison-Wesley, Boston USA

Unit 2

Required readings and other resources:

- Ariadne Training (2001), "UML Applied Object Oriented Analysis and Design Using the UML", Ariadne Training Limited
- Larman C. (2004), "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", (3rd Edition) 3rd Edition, Prentice Hall; 3 edition (October 30, 2004), ISBN-13: 978-0131489066
- Liu Z., (2001), "Object-Otiented Software Development Using UML", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.
- Ojo A. and Estevez E., (2005), "Object-Oriented Analysis and Design with UML", Training Course, The United Nations University, UNU-IIST International Institute fo
- Software Technology, e-Macao Report 19, Version 1.0, October.
- Booch G., Rumbaugh J. and Jacobson I. (1998), " Unified Modeling Language User Guide", Addison Wesley , First Edition October 20, 1998 , ISBN: 0-201-57168-4, 512 pages
- Sommerville Ian (2000), "Software Engineering (6th Edition)". Addison-Wesley, Boston USA

Unit 3

Required readings and other resources:

- Ariadne Training (2001), "UML Applied Object Oriented Analysis and Design Using the UML", Ariadne Training Limited
- Larman C. (2004), "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", (3rd Edition) 3rd Edition, Prentice Hall; 3 edition (October 30, 2004), ISBN-13: 978-0131489066
- Liu Z., (2001), "Object-Otiented Software Development Using UML", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.
- Ojo A. and Estevez El., (2005), "Object-Oriented Analysis and Design with UML", Training Course, The United Nations University, UNU-IIST International Institute for

Software Technology, e-Macao Report 19, Version 1.0, October.

- Pressman Roger S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783
- Sommerville Ian (2000), "Software Engineering (6th Edition)". Addison-Wesley, Boston USA
- Pressman Roger S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783
- Sommerville Ian (2000), "Software Engineering (6th Edition)". Addison-Wesley, Boston USA

Unit 4

Required readings and other resources:

- Larman C. (2004), "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", (3rd Edition) 3rd Edition, Prentice Hall; 3 edition (October 30, 2004), ISBN-13: 978-0131489066
- Liu Z., (2001), "Object-Otiented Software Development Using UML", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.
- Ojo A. and Estevez El., (2005), "Object-Oriented Analysis and Design with UML", Training Course, The United Nations University, UNU-IIST International Institute for Software Technology, e-Macao Report 19, Version 1.0, October.

Unit 5

Required readings and other resources:

- Liu Z., (2001), "Object-Otiented Software Development Using UML", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.
- Larman C. (2004), "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", (3rd Edition) 3rd Edition, Prentice Hall; 3 edition (October 30, 2004), ISBN-13: 978-0131489066
- Ojo A. and Estevez El., (2005), "Object-Oriented Analysis and Design with UML", Training Course, The United Nations University, UNU-IIST International Institute for Software Technology, e-Macao Report 19, Version 1.0, October.
- Pressman Roger S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783
- Sommerville Ian (2000), "Software Engineering (6th Edition)". Addison-Wesley, Boston USA

Unit 0 - Basic Knowledge

Unit Introduction

The unit reviews software development life cycles (SDLC) or software development process. The unit is highlighting more on the phases to be used in this module. At the end, the unit is providing a number of assessments for student to answer.

Unit Objectives

Upon completion of this unit you should be able to:

1. Describe the software development process term
2. Explain a set of activities in software development process

KEY TERMS

Software process:

A software process is a set of activities and associated results. When those activities are performed in specific sequence in accordance with ordering constraints, the desired results are produced.

Software development process:

A software development process is often described in terms of a set of activities needed to transform a user's requirements into a software system.

SDLC:

System development life cycle (SDLC) means combination of various activities. In other words we can say that various activities put together are referred as system development life cycle

Software Development

Introduction

The software-development process or sometimes known as software development life-cycle (SDLC) is used to facilitate the development of a large software product in a systematic, well-defined, and cost-effective way. An information system goes through a series of phases.

This activity presents the review on software development process.

Software development Process

The software industry considers software development as a process. According to Liu, 2001, "A process defines who is doing what, when and how to reach a certain goal". Software engineering is a field, which combines processes, methods, and tools for the development of software. The concept of process is the main step in the software engineering approach, thus, a software process is a set of activities and associated results. When those activities are performed in specific sequence in accordance with ordering constraints, the desired results are produced.

There are four fundamental process activities which are common to all software processes:

a. Software specifications: The functionality of the software and constraints on its operation must be defined.

b. Software development: Software that meets the specifications must be produced.

c. Software validation: The software must be validated (confirmed) to ensure that it does what the customer wants.

d. Software evolution: The software must evolve to meet changing customer needs.

A process to build a software product or to enhance an existing one is called a software development process. A software development process as depicted in Figure 0.1 is thus often described in terms of a set of activities needed to transform a user's requirements into a software system. The client's requirements define the goal of the software development. They are prepared by the client (sometime with the help of a software engineer) to set out the services that the system is expected to provide, i.e. functional requirements.

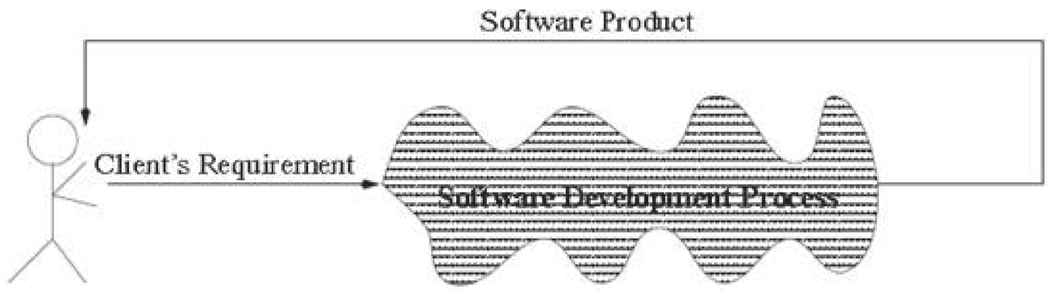


Figure 0.1: A View of the Software Development Process (Source: Liu, 2001)

The functional requirements should state what the system should do rather than how it is done.

Apart from functional requirements, a client may also have non-functional constraints that s/he would like to place on the system, such as the required response time or the use of a specific language standard.

We must bear in mind about the following facts which make the requirement capture and analysis very difficult:

a. The requirements are often incomplete.

b. The client's requirements are usually described in terms of concepts, objects and terminology that may not be directly understandable to software engineers.

c. The client's requirements are usually unstructured and they are not rigorous (exact), with repetitions, redundancy, vagueness, and inconsistency.

d. The requirements may not be feasible.

Therefore, any development process must start with the activities of capturing and analyzing the client's requirements.

Software Development Life Cycle

Software development life cycle (SDLC) has various activities. When various activities are put together, are referred to as system development life cycle or software development process. The phases in the software development life cycle can be identified by different names and there are rules regarding what are to be included. Each activity has its own output. Activities or phases to be adopted in object oriented analysis and design course include:

1. Requirement Capture and Analysis
2. Software Design
3. Implementation/Coding and Unit Testing

Requirement Capture and Analysis

Requirement Elicitation/Gathering is a communication process between the parties involved and affected in the problem situation. The tools in elicitation are meetings, interviews, video conferencing, e-mails, and existing documents study and facts findings. The sources are: Customer (Initiator), End Users, Primary Users, Secondary Users, Stakeholders.

Requirement analysis is a very important and essential activity after elicitation. In this phase, each requirement is analyzed from the point-of-view of validity, consistency, and feasibility for firm consideration in the Requirement Definition Document (RDD) and then in the Software Requirements Specification (SRS). Analysis emphasizes an investigation of the problem and requirements, rather than a solution

This requirements definition document describes what the customer would like to see, including:

1. Outline of the general purpose of the system
2. Describe the background and objectives of system development. For example, if a system is to replace an existing approach, we explain why the existing system is unsatisfactory
3. Outline a description of the approach. Note that the purpose of the requirements document is to discuss the problem, not the solution; the focus should be on how the system is to meet the customer's needs.
4. Describe the detailed characteristics of the proposed system. Define the system boundaries and interfaces across it. The system functions are also explained.
5. Discuss the environment in which the system will operate. Include requirements for support, security, and privacy and any special hardware or software constraints should be addressed.

This portion of the analysis confirms the place of the requirements in RDD on its own and along with others. The SRS is a specification for a particular software product, program, or set of programs that perform certain functions in a specific environment. The second portion of analysis attempts to find for each requirement; its functionality, features, and facilities and the need for these under different conditions and constraints. The purpose of the requirement capture and analysis is to aim the development toward the right system. Its goal is to produce a document called requirement specification. The whole scope of requirement capture and analysis forms the so-called requirement engineering. The process activities include:

1. Domain understanding: this is to understand the application domain. The concept is explored and the client's requirements are elicited.
2. Requirements capture or collection: This is the process of interacting with stakeholders in the system to discover their requirements. More domain understanding develops further during this activity.
3. Classification: This activity takes the unstructured collection of requirements captured in the earlier phase and organizes them into coherent clusters, and then prioritizes the requirements according to their importance to the clients and the users.

4. Validation: This is to check if the requirements are consistent and complete, and to resolve conflicts between requirements.
5. Feasibility study: This is to estimate whether the identified requirements may be satisfied using the software and hardware technologies, and to decide if the proposed system will be cost-effective.

Software Design

Design is a meaningful representation of something that is to be built. According to Agarwal et al (2010), the term design describes the final system and process by which it is developed. Software design involves taking a software requirement specification obtained in requirement capture and analysis and adding details needed for actual implementation in a computer.

Input includes an understanding of the following: Requirements, Environmental constraints and Design criteria. After the specification is produced through requirement analysis, the requirement specification undergoes two consecutive design processes.

Architectural (or logical) design

The requirements are partitioned into components. Large systems are always decomposed into subsystems that provide some related set of services. The initial design process of identifying these subsystems and establishing a framework for subsystem control and communication is called architectural design. Architectural design represents the structure of data and program components that are required to build a computer-based system.

Software design results in an architectural design document which describes what each component must do and how they interact with each other to provide the overall required services.

Detailed (or Physical) Design

Then each component in turn is designed; this process is termed detailed (or physical design). The detailed design document describes how each component does what it is required to do and thus how the whole system does what it is required to do.

The design process translates requirements into a representation of the software that can be assessed for quality before coding begins. Like requirements, the design is documented and becomes part of the software configuration. The activities of the design process and the associated results are depicted in Figure 0.2.

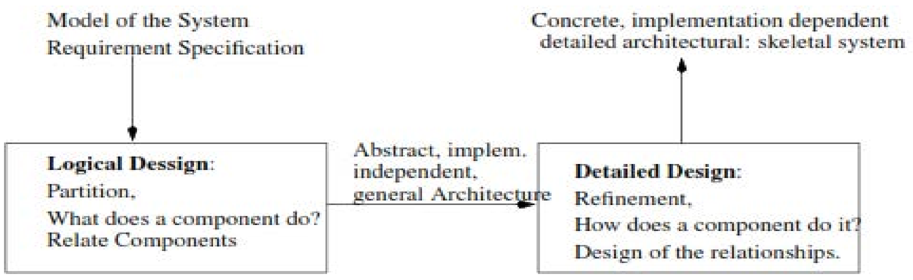

Model of the System
Requirement Specification
Concrete, implementation dependent
detailed architectural: skeletal system
Logical Dessign:
Partition,
What does a component do?
Relate Components
Abstract, implem. independent, general Architecture
Detailed Design:
Refinement,
How does a component do it?
Design of the relationships.

Figure 0.2: A view of the design process

Implementation/Coding and Unit Testing

During this stage, each of the components from the design is realized as a program unit. Each unit then must be either verified or tested against its specification obtained in the design stage. The goal of the coding phase is to translate the design of the system into code in a given programming language. In this phase the aim is to implement the design in the best possible manner. Testing is the major quality-control measure used during software development. Its basic function is to detect errors in the software. Thus, the goal of testing is to uncover requirement, design, and coding errors in the program. This process is depicted in Figure 0.3.

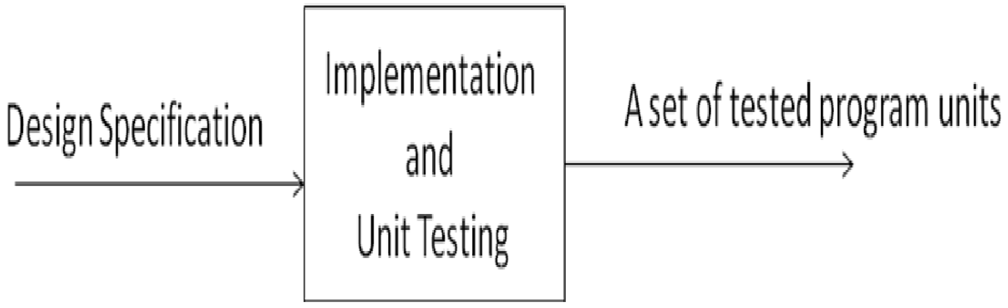

Design Specification
Implementation and Unit Testing
A set of tested program units

Figure 0.3: A View of the Implementation and Unit Testing Process

Conclusion

Software systems are built successfully by applying a certain adaptable process. This process leads to a high-quality product that meets the users' requirements, who are the ones who will use the product. This is software engineering approach to be applied in a discipline manner.

Activity Assessment

a. Define the following terms as applied to software development process:

i. SDLC

ii. System design

b. Explain the basic phases in the software-development life-cycle.

c. List and describe four fundamental process activities which are common to all software processes

d. Explain why do we need a software development process?

e. Find more about software development process, especially the requirements that a good software development process must be met.

UNIT SUMMARY

For many years, the term object oriented (OO) was used to denote a software development approach that used one of a number of object-oriented programming languages (e.g., Java, C++). Today, the OO paradigm encompasses a complete view of software engineering; hence object-oriented approach has to be systematic, disciplined in following the software development process or software development life cycle.

Unit Assessment

<u>Instructions</u>

Answer the following questions.

i. Describe the following terms: (Answer: section 0.1.2)

(a) Process

(b) Software Development Process

(c) Software validation

(d) Software implementation (Answer: section 0.1.3.3)

ii. List four facts which indicate that the requirement capture and analysis process to be very difficulty(Answer: section 0.1.2)

iii. List and Describe four fundamental process activities which are common to all software development processes (Answer: Section 0.1.3.1)

iv. Differentiate between Requirement Definition Document and Software Requirement Specification (Answer: section 0.1.3.1)

v. List information needed to be included in requirement Definition Document (Answer: section 0.1.3.1)

vi. Differentiate between the following terms:

(a) Software analysis and software design (Answer: section 0.1.3.1 and 0.1.3.2)

(b) Logical design and physical design (Answer: section 0.1.3.2)

vii. Define the term "classification" as applied in requirement analysis (Answer: section 0.1.3.1)

Answers

i. Describe the following terms: (3 marks)

(a) Software Development Process

A software development process is a set of activities and associated results. When those activities are performed in specific sequence in accordance with ordering constraints, the desired results are produced.

(b) Software validation

If the process of confirming or ensuring if the software developed does what the customer wants.

(c) Software implementation

Software implementation is a coding phase aiming to translate the design of the system into code in a given programming language. In this phase the aim is to implement the design in the best possible manner.

ii. List four facts which indicate that the requirement capture and analysis process to be very difficult. (4 marks)

(a) The requirements are often incomplete.

(b) The client's requirements are usually described in terms of concepts, objects and terminology that may not be directly understandable to software engineers.

(c) The client's requirements are usually unstructured and they are not rigorous (exact), with repetitions, redundancy, vagueness, and inconsistency.

(d) The requirements may not be feasible.

iii. List and Describe process four fundamental process activities which are common to all software processes (6 marks)

(a) Software specifications: The functionality of the software and constraints on its operation must be defined.

(b) Software development: Software that meets the specifications must be produced.

(c) Software validation: The software must be validated (confirmed) to ensure that it does what the customer wants.

(d) Software evolution: The software must evolve to meet changing customer needs.

iv. Differentiate between Requirement Definition Document and Software Requirement Specification (4 marks)

Requirement Definition Document (RDD)

RDD is the document which describes what the customer would like to see. The information is provided by the customer, however developer can assist with the help of meetings, interviews, video conferencing, e-mails, and existing documents study and facts findings.

Software Requirement Specification

The SRS is a specification for a particular software product, program, or set of programs that perform certain functions in a specific environment. The analysis attempts to find for each requirement; its functionality, features, and facilities and the need for these under different conditions and constraints.

v. List information needed to be included in requirement Definition Document (5 marks)

Information includes:

(a) Outline of the general purpose of the system

(b) Describe the background and objectives of system development.

(c) Outline a description of the approach..

(d) Describe the detailed characteristics of the proposed system. Define the system boundaries and interfaces across it. The system functions are also explained.

(e) Discuss the environment in which the system will operate. Include requirements for support, security, and privacy and any special hardware or software constraints should be addressed.

vi. Differentiate between the following terms:

(a) Software analysis and software design (4 marks)

Software analysis

Software analysis is the process of analyzing requirement analysis. In this phase, each requirement is analyzed from the point-of-view of validity, consistency, and feasibility for firm consideration in the Requirement Definition Document (RDD) and then in the Software Requirements Specification (SRS). Analysis emphasizes an investigation of the problem and requirements, rather than a solution

Software design

Design is a meaningful representation of something that is to be built. The term design describes the final system and process by which it is developed. Software design involves taking a software requirement specification obtained in requirement capture and analysis and adding details needed for actual implementation in a computer. The design process translates requirements into a representation of the software that can be assessed for quality before coding begins

(b) Logical design and physical design (Answer: section 0.1.3.2)

Architectural (or logical) design

The requirements are partitioned into components. Large systems are always decomposed into subsystems that provide some related set of services. The initial design process of identifying these subsystems and establishing a framework for subsystem control and communication is called architectural design. Architectural design represents the structure of data and program components that are required to build a computer-based system.

Software design results in an architectural design document which describes what each component must do and how they interact with each other to provide the overall required services.

Detailed (or Physical) Design

Then each component in turn is designed; this process is termed detailed (or physical design). The detailed design document describes how each component does what it is required to do and thus how the whole system does what it is required to do.

vii. Define the term "classification" as applied in requirement analysis (Answer: section 0.1.3.1)

Classification is an activity of taking the unstructured collection of requirements captured in the earlier phase and organize them into coherent clusters, and then prioritize the requirements according to their importance to the clients and the users.

Unit Readings and Other Resources

- Liu, Z. (2001), "Object-Oriented Software Development Using UML", March, The United University – International Institute for Software Technology (UNU/IIST), Report No. 229.
- Nellen, T. and Mayo, L. (2000), "We Learn by Doing", URL: http://english.ttu.edu/kairos/5.1/coverweb/nellenmayo/doing.html.
- Bjork R. C., (2004), "ATM Simulation", ATM Online", URL: http://www.math-cs.gordon.edu/courses/cps211/ATMExample/

Unit 1 - Principles of Object Oriented

Unit Introduction

The unit starts by emphasizes the importance of learning-by-doing so as to grasp the Object-oriented approach to software development. In this, a number of case projects have been given; two for a whole class and others for group of student.

The unit also compares software development methodologies, mainly structured and object-oriented approaches. The object oriented principles: Abstraction, Hierarchy, Modularity, and Encapsulation: and fundamental concepts of Object Orientation: Objects, Classes, Inheritance and Polymorphism are presented. This unit explains associations and links: Association, Composition, Inheritance or Generalization, Aggregation. It explains Super Class, Sub Class, and Multiplicity.

Unit Objectives

Upon completion of this unit you should be able to:

1. Describe the overview of the case projects to be used in practicing the object-oriented approach
2. Differentiate between structured approach and object-oriented approach
3. Explain the basic principles of object-oriented
4. Describe fundamental concepts of Object Orientation
5. Demonstrate the use of associations and links

KEY TERMS

Structured Software Development Approach:

Structured approach is to look at the problem, and then design a collection of functions that can carry out the required tasks. If these functions are too large, then the functions are broken down until they are small enough to handle and understand.

Object-Oriented Software Development Approach:

The strategy in the OO software development is to view the world as a set of objects. They interact with and collaborate with each other to provide some higher level behaviour.

Class:

A class is a 00 concept that encapsulates the data and procedural abstractions required to describe the content and behavior of some real-world entity.

Object:

An object is the basic run-time entity in an object-oriented system. On object can represent a person, a location, an account, a table of data or any item that the program must handle. A programming problem is analyzed in terms of objects and the nature of communication between them.

Learning Activities

Activity 1 - Case Studies

Introduction

The course will have one general case study namely: Point-of-Sale terminal" by Liu (2001) which will be used to demonstrate object oriented approach in software development. The course will again use a Bank Automatic Teller Machine (ATM) system to strengthen explanation in other perspective whenever needed. Students are advised to have their own case study projects to do the same so that they can grasp easily the approach by doing.

The stated case study problems (POST and ATM) have been chosen because they are familiar to many people, yet rich with complexity and interesting design problems. That allows us to concentrate on learning fundamental OOA/D, requirements analysis, UML and patterns, rather than explaining the problems.

Course Case Study One: Point-of-Sale Terminal (POST)

The first case study is the NextGen point-of-sale (POS) system. In this apparently straight forward problem domain, we shall see that there are interesting requirement and design problems to solve. In addition, it's a real problem—groups really do develop POS systems with object technologies. The course will use a "Point-of-Sale Terminal" case study as being used by Liu (2001) as follows:

A point-of-sale terminal (POST) is a computerized system used to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and a bar code scanner, and software to run the system (See Figure 0.4). It interfaces to various service applications, such as a third-party tax calculator and inventory control.

These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).

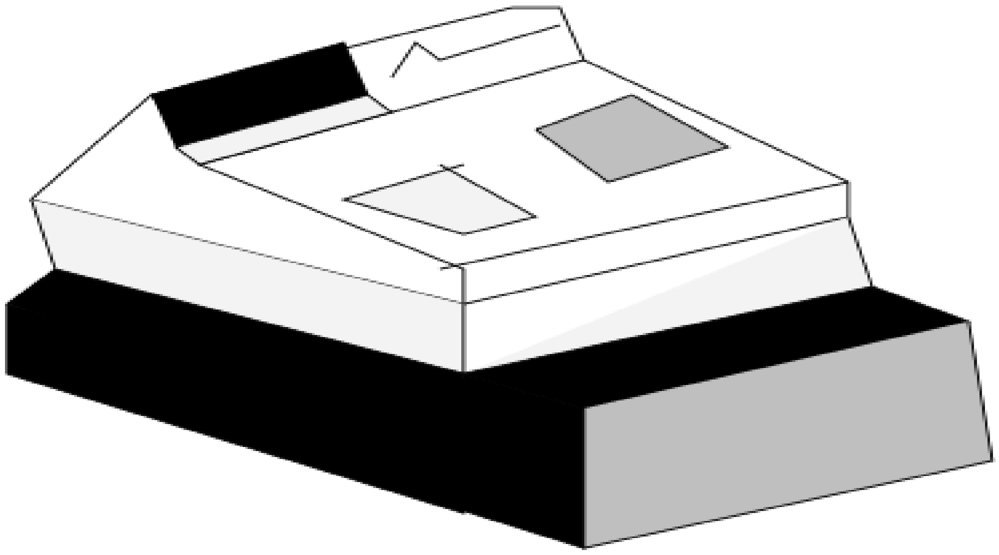


Figure 1.1: Point-of-Sale Terminal

A POST system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input and wireless PDAs.

Furthermore, we are creating a commercial POST system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

Assume that we have been requested to create the software to run a point-of-sale terminal. Using an object-oriented development strategy, we are going to proceed through the requirement, object-oriented analysis, design, and implementation.

Case Study Two: Bank Automatic Teller Machine (ATM)

To support the POST case study, the course will also reference on the commonly used Bank Automatic Teller Machine (ATM) by Bjork R. C., (2004). The aim is to strengthen understanding of the OO approach in different scenarios.

Though ATM are being modified daily to accommodate functionalities required by customers, the scenario to be followed will be as follows:

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN) - both of which will be sent to the bank for validation as part of each transaction. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned.

This case study will be used to show examples for the different models created during the development phases. In each model we present a particular aspect of this case study.

Students' Case Projects

A group of three up to five students may use one of the following projects to practice the use of object oriented approach in software development. Students are not bound to use the listed projects below; instead, they can find their own simple projects of interest to do the same under the supervision of their supervisors. This is important as per the learn-by-doing advocates. All students' case studies should use Object-Oriented system analysis and design approach. These projects are going to be referenced in exercise activities given as a guide on what to do in each phase of software development process.

Project 1:

Design and implement a Web-based system to record student registration and grade information for courses at a university.

Project 2:

Design and implement a system that permits recording of course performance information specifically, the marks given to each student in each assignment or exam of a course, and computation of a (weighted) sum of marks to get the total course marks. The number of assignments/exams should not be predefined; that is, more assignments/exams can be added at any time. The system should also support grading, permitting cutoffs to be specified for various grades.

You may also wish to integrate it with the student registration system of Project 0.1 (perhaps being implemented by another project team).

Project 3:

Design and implement a Web-based system for booking classrooms at your university. Periodic booking (fixed days/times each week for a whole semester) must be supported. Cancellation of specific lectures in a periodic booking should also be supported.

You may also wish to integrate it with the student registration system of Project 0.1 (perhaps being implemented by another project team) so that classrooms can be booked for courses, and cancellations of a lecture or extra lectures can be noted at a single interface, and will be reflected in the classroom booking and communicated to students via e-mail.

Conclusion

The easiest way to learn a new area, especially software development, is by employing the "learn-by-doing" technique. Nellen and Mayo (2000) emphasize this by saying that if knowledge is power, then we gain that knowledge by doing. In order to understand the object-oriented approach in software development, it is very important to practice the development through case studies.

Activity Assessment

Select one project out of students' case projects given in sub section 1.1.3 or come up with your own project title of interest and explain an overview of it in a form of an abstract.

Activity 2 – Software development

Introduction

There are several software development methodologies employed. The two most commonly applied approaches are: structured approach and object-oriented approach. Before discussing these two approaches it is important to have an understanding of the similar terminologies as refereed to computer programming.

Procedural-Oriented Programming

Procedural-oriented programmings are conversional programming using high-level languages such as C, FORTRAN, Pascal, etc. In procedural-oriented approach, the problem is reviewed as a sequence of things to be done. Each statement in the language tells the computer to do something: get some input, add these numbers, multiply, display the output, etc.

A program in a procedural-oriented language is a list of instructions. In procedural oriented languages the primary focus is on functions, we write a list of instructions for computer to follow, and organize these instructions into groups known as functions. A number of functions are written to accomplish different tasks. In functions, very little attention is given to the data that are being used by various functions. In this case, global data is more vulnerable to an inadvertent (unintentional) change by a function. In large programs, it is very difficult to identify what data is used by which function and in case we need to check a data structure, we need to check all functions that access that data and this provides a chance for bugs to creep in. Few characteristics of procedural languages are:

- Emphasis is on doing tasks
- Large programs are broken down into smaller programs known as functions
- Most of the time, functions share global data
- Data move openly around the system from function to function
- It employs top-down approach in program design

Object-Oriented Programming

The main feature of object-oriented programming (OOP), like C++, Java, is that its main emphasis is on data rather than functions or procedures. In object-oriented programming data is treated as a critical element in the program and does not allow it to flow freely around the system. It ties data to the functions that work on it and protects it from accidental modifications from outside functions. In procedural-oriented programming languages the problem is broken into functions but OOP allows decomposing a problem into a number of entities called objects and then building data and functions around these objects. The data of an object can be accessed only by the function associated with that object. However, functions of one object can access the functions of other objects. Few characteristics of object-oriented languages are:

- Emphasis is on data rather than functions
- Programs are divided into objects
- Data structures are designed such that they characterize the objects
- Data is hidden and cannot be accessed by external functions
- Objects can communicate with one another through functions
- OOP follows bottom-up approach in program design

Software Development Methodologies – Structured Approach

Structured approach is the most widely used software development methodology. Techniques which are invariably present, such as entity relationship diagrams, dataflow diagrams and data dictionaries are also clearly mentioned.

Structured Systems Analysis and Design as described by Ariadne Training (2001), prescribe analyzing and designing software systems through functional decomposition – i.e. examining an information system in terms of the functions it performs and the data it uses and maintains. Like procedure-oriented programming, the analyst identifies the major functions or processes of a system, then breaks or decomposes each function down into its smaller composite steps. As an example for a college management system, the system holds the details of every student and tutor in the college.

It can store information about courses and track which student is following which course. A possible functional design would be:

- "add_student",
- "enter_for_exam",
- "check_exam_marks",
- "issue_certificate",
- "expel_student"

Structured approach is generally called the structured or procedural or traditional systems development. As emphasized by Liu, 2001, structured or sometimes called "Functional – Oriented" method, until the middle of the 1990s, most of software engineers were used to the top-down functional design methods, whose defining aspects include:

- It is strongly influenced by the programming languages such as ALGOL, Pascal and C, all of which offer routines as their highest-level abstractions.
- The functions of a software system are considered as the primary criterion for its decomposition.
- It separates the functions and data, where functions, in principle, are active and have behaviour, and data is a passive holder of information, which is affected by functions.

The top-down decomposition typically breaks the system down into functions, whereas data is sent between those functions. The functions are broken down further and eventually converted into source code as shown in Figure 1.1.

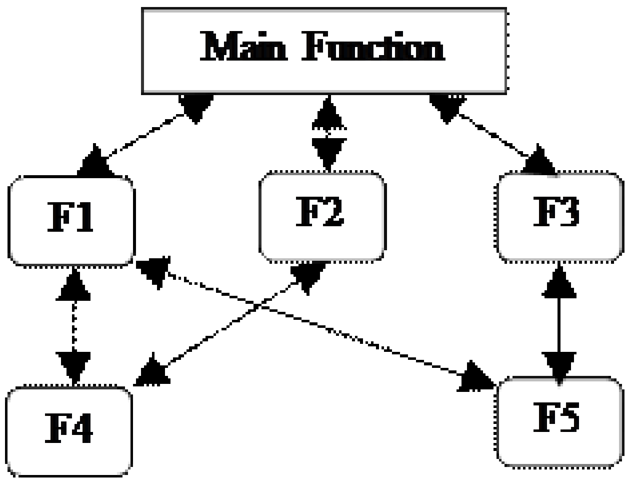


Figure 1.2: Functional, Top-Down Decomposition (Source Liu, 2001)

The software developers found the following problems with the functional technology (Liu, 2001):

- Products developed with this approach were difficult to maintain. This was mainly because that all functions share a large amount of data, and they must know how the data are stored. To change a data structure, there was a need to modify all the functions relating to the structure.
- The development process was not stable as changes in the requirements will be mainly reflected in its functionality. Therefore, it was difficult to retain the original design structure when the system evolves.
- The development process gets into the "how" business too soon, as when decomposing a function into sub functions, it cannot avoid from talking about first do this and then do that, and so on.
- This approach only captures the "part-of" abstraction.
- Obviously, this approach does not support programming in object-oriented languages, such as Smalltalk, Eiffel, C++, and Java.

Systems Development Methodologies – Object Oriented Approach

Object-oriented (OO) approach has been identified as the new paradigm for system development. In contrast to structured approach and in similar to object-oriented programming, the object oriented system development decomposes the system down into a set of objects – i.e. it examines the system in terms of the things in the system and how these things act and interrelate with each other to provide some higher level behaviour. In object-oriented approach:

- The analyst first identifies the objects that comprise the system, then
- They create an object model which groups the objects into classes, and describes each class in terms of its attributes (or data), methods (or functions), and relationships to other classes.

Object-oriented technology

Object Orientation is about viewing and modeling the world/system as a set of interacting and interrelated objects. Objects have the following characteristics (see Figure 1.2):

- An object is simply a tangible entity in the real world (at the requirement analysis phase) or represents a system entity (at the design stage).
- Objects are responsible for managing their own private state, and offering services to other objects when is it requested. Thus, data and functions are encapsulated within an object.

- The system's functionality is observed by looking at how services are requested and provided among the objects, without the need to get into the internal state changes of the objects.
- Objects are classified into classes, and objects belonging to the same class have common properties, such as attributes and operations.

OO approach reflects the "is a" abstraction as well as the "part of" abstraction better compared to structured approach which is more for "part of" abstraction.

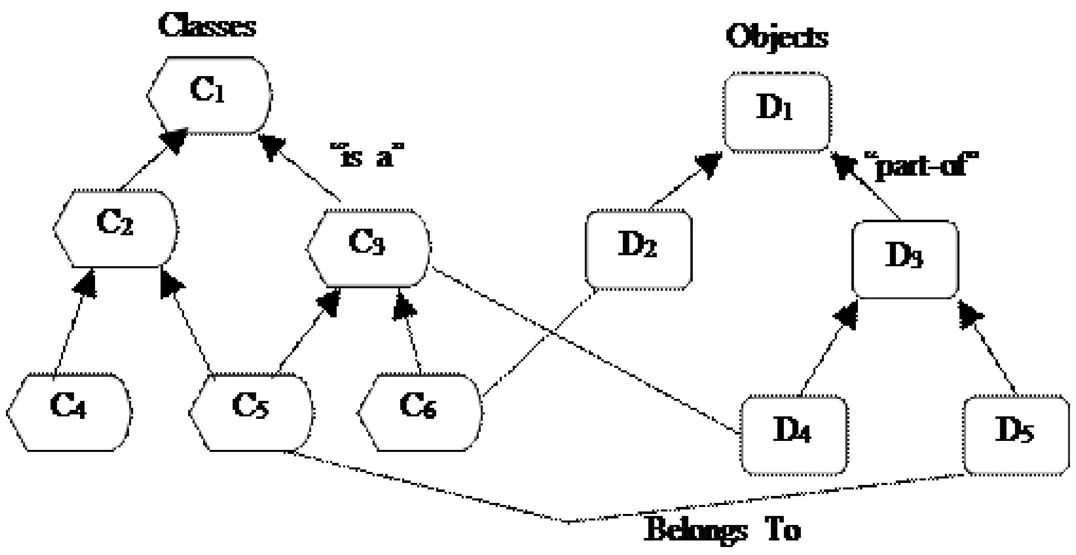


Figure 1.3: A Canonical Form of a Complex System (Source Liu, 2001)

Structured Approach versus OO Approach

Although both paradigms decompose a system:

- The traditional structured approach focuses on what a system does i.e. its major functions; or the verbs that describe the system; Object oriented systems analysis and design (OOA&D) focuses on what the system is composed of, i.e. its major players or objects, or the nouns that describe the system
- In structured analysis, the analyst is concerned with the functions performed on the data; in OOA&D, the analyst is concerned with objects performing functions.
- Both are concerned with objects and data, but the structural analyst separates them into functions performed by objects on the data, while the OO analyst combines them into an object which performs functions on the data.

For many years, the term object oriented (OO) was used to denote a software development approach that used one of a number of object-oriented programming languages (e.g., Java, C++). Today, the OO paradigm encompasses a complete view of software engineering. This course will give attention to software engineering using OO approach.

Conclusion

Object oriented approach differs to structured approach since it combines data and behaviour into a class. Programs create instances of the class known as objects. Objects can collaborate with each other by calling each other's methods. The data in an object is encapsulated and only the object itself can modify the data.

Activity Assessment

i. List two basic differences between traditional systems analysis and object-oriented analysis.

ii. Object-oriented development methods are rapidly replacing older structured development methods. Has structured development failed and why should object oriented development prove to be any better?

Activity 3 –Principles and Concepts of Object Oriented

Introduction

Object-oriented programming is based on a number of independent self-contained objects working together to perform a function. This kind of approach to developing code can give a number of benefits, such as easier code maintenance and enhanced re-usability. This activity introduces the principles used in object-oriented. These include: abstraction, encapsulation, modularity and Hierarchy. It also presents the fundamental concepts used in object-oriented. These include: objects, classes, inheritance, and polymorphism

Principles of Object Oriented

Principle 1 - Abstraction

Abstraction is a model that includes most important aspects of a given system while ignoring less important details. The skill of abstraction is important to capture the essences and ignore the non-essences. Object oriented is a better abstraction of the Real World, this means that if the problem changes (i.e. the requirements change, as they always do), the solution should be easier to modify, as the mapping between the problem and solution is easier.

Ojo and Estevez, (2015) state that abstraction is the process allowing to focus on most important aspects while ignoring less important details. It allows management of complexity by concentrating on essential characteristics that makes an entity different from others. Figure 1.3 shows an example of an order processing abstraction.

Principle 2 - Encapsulation

In encapsulation, only the instance that owns an item of data is allowed to modify or read it. The object chair (and all objects in general) encapsulates data (the attribute values that define the chair), operations (the actions that are applied to change the attributes of chair), other objects (composite objects can be defined), constants (set values), and other related information. Encapsulation means that all of this information is packaged under one name and can be reused as one specification or program component. In other words, encapsulation separates implementation from users or clients (Ojo and Estevez, 2015). Encapsulation requirements include:

- To expose the purpose of an object
- To expose the interfaces of an object
- To hide the implementation that provides behaviour through interfaces
- To hide the data within an object that defines its structure and supports its behaviour
- To hide the data within an object that tracks its state

Advantages of encapsulation include:

- Facilitate separation of an interface from implementation, so that one interface may have multiple implementations
- Data held within one object cannot be corrupted by other objects

Principle 3 - Modularity

Modularity deals with the process of breaking up complex systems into small, self contained pieces that can be managed easily. Modularity can be done by:

- Decomposing the problem into smaller sub-problems
- Trying to solve each sub-problem separately Each solution is a separate component that includes
- Interface: types and operations visible to the outside
- Specification: intended behavior and property of interface
- Implementation: data structures and functions hidden from outside

See order processing system in Figure 1.5 as an example of modularity.

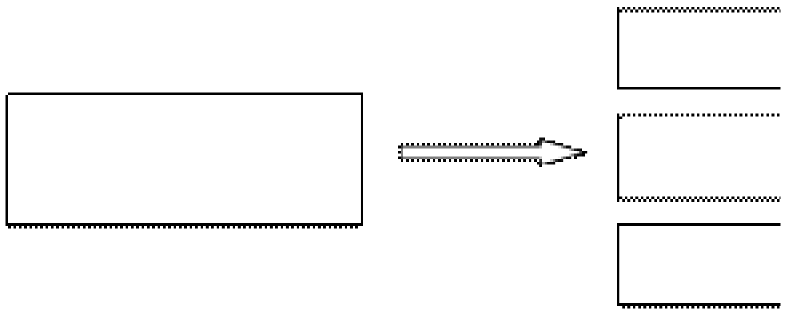


Figure 1.6: Order Processing System

Principle 4 - Hierarchy

Hierarchy is an ordering of abstractions into a tree like structure. Complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached. Figure 1.6 shows an example of the Hierarchy object oriented principle.

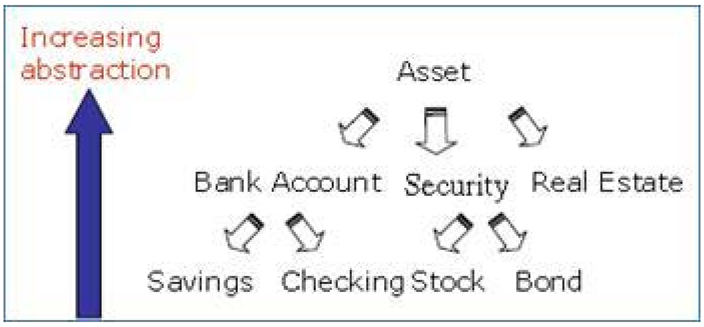


Figure 1.7: Example of Hierarchy

<u>Object Oriented Concepts</u>

OO Concept 1 - Objects

Any discussion of object-oriented software engineering must begin by addressing the term object-oriented. What is an object-oriented viewpoint? Why is a method considered to be object-oriented? What is an object? As defined by Ojo and Estevez (2015), Object Orientation is about viewing and modeling the world/system as a set of interacting and interrelated objects. Features of the OO approach:

a. The universe consists of interacting objects

b. Describes and builds systems consisting of objects.

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a location, an account, a table of data or any item that the program must handle. They may also represent user-defined data such as vectors, time and lists. A programming problem is analyzed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects.

When a program is executed, the objects interact by sending messages to one another. For example, if "student" and "teacher" are two objects in a program, then the student object may send a message to the teacher object requesting for his marks scores in a test. Each object contains data and code to manipulate the data. Objects can interact without having knowing-details of one another's data or code.

Consider an example of a real world object. Chair is a member (the term instance is also used) of a much larger class of objects that we call furniture. A set of generic attributes can be associated with every object in the class furniture (For example, all furniture has a cost, dimensions, weight, location, and color, among many possible attributes). These apply whether we are talking about a table or a chair, or a sofa. Because chair is a member of furniture, chair inherits all attributes defined for the class as shown in Figure 1.8. Once the class has been defined, the attributes can be reused when new instances (objects) of the class are created.

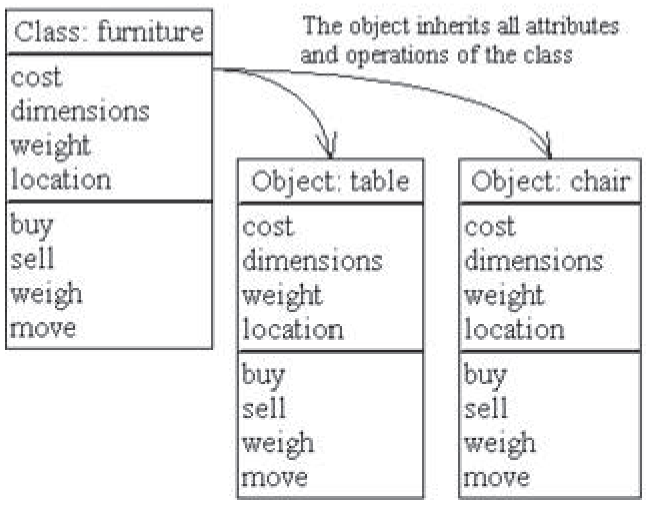


Figure 1.9: Chair and Table are Objects of Furniture

Objects in the real world can be characterized by two things: state and Operations. Each real world object has data and behavior. The data for an object are generally called the Attributes of the object. The different behaviours of an object are called the Methods (or operations) of the object

State is the collection of information held (i.e., stored) by the object. It can change over time. It changes as the result of an operation performed on the object. It cannot change spontaneously. The state is encapsulated within the object is not directly visible.

The various components of the state are sometimes called the attributes of the object. Operation is a procedure that takes the state of the object and zero or more arguments and changes the state and/or returns one or more values. Objects permit certain operations and not others.

OO Concept 2 - Classes

The term Class is simply a template for an object. A class is a description of a set of related objects that share the same attributes, operations. A class describes what attributes and methods will exist for all instances of the class. A class is just a description; it doesn't really exist as such until you declare an instance of the class relationships, and semantics.

Objects contain data and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class. In fact, objects are variables of type class. Once a class has been defined, we can create any number of objects belonging to that class. Objects are created using class definitions as templates. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type. Graphically, a class is rendered as a rectangle. This is an important concept to grasp - the class you create can't do anything until you use it.

A Class Name must be unique within its enclosing package. Every class must have a name that distinguishes it from other classes. A name is a textual string. That name alone is known as a simple name; a path name is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name as shown in Figure 1.9 as examples.

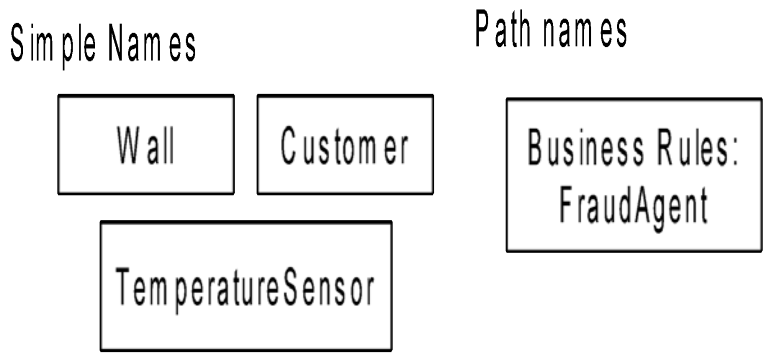


Figure 1.10: Examples of Class Names (Source: Ojo and Estevez, 2005)

An attribute is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class. An attribute name may be text. In practice, an attribute name is a short noun or noun phrase that represents so.me property of its enclosing class. Typically, you capitalize the first letter of every word in an attribute name except the first letter, as in name or loadBearing..

An operation is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class. A class may have any number of operations or no operations at all. An operation name may be text. In practice, an operation name is a short verb or verb phrase that represents some behavior of its enclosing class. Typically, you capitalize the first letter of every word in an operation name except the first letter, as in move or isEmpty.

An attribute is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class. An attribute name may be text. In practice, an attribute name is a short noun or noun phrase that represents so.me property of its enclosing class. Typically, you capitalize the first letter of every word in an attribute name except the first letter, as in name or loadBearing..

An operation is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class. A class may have any number of operations or no operations at all. An operation name may be text. In practice, an operation name is a short verb or verb phrase that represents some behavior of its enclosing class. Typically, you capitalize the first letter of every word in an operation name except the first letter, as in move or isEmpty.

OO Concepts 3 – Inheritance

Inheritance is one of the key differentiators between conventional (structured) and OO systems. A subclass Y inherits all of the attributes and operations associated with its superclass, X. This means that all data structures and algorithms originally designed and implemented for X are immediately available for Y—no further work need be done. Reuse has been accomplished directly. Any change to the data or operations contained within a superclass is immediately inherited by all subclasses that have inherited from the superclass. Therefore, the class hierarchy becomes a mechanism through which changes (at high levels) can be immediately propagated through a system.

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification.

For example, the grandchild is a part of the class parent which is again a part of the class grandparent. In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. In C++, the original class is called a base class and the class which acquired the features of base class is called derived class. The new class will have the combined features of both the classes. The real and power of the inheritance mechanism is that it allows the programmer to reuse a class. That is inheritance encourages sharing and reuse of both design information and program code.

OO Concept 4 - Polymorphism

Polymorphism is another important OOP concept. Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data used in operation. For example, consider the operation of addition.

- For two numbers, the operation will generate a sum
- If the operands are strings, then the operation would produce a third string by concatenation

This means that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context.

Derived classes can redefine the implementation of a method. Example 1:

- Consider a class "Transport"
- One method contained in transport must be move(), because all transport must be able to move
- If we wanted to create a Boat and a Car class, we would certainly want to inherit from the transport class, as all Boats can move and all Car can move, however in different ways

Conclusion

In software engineering, the models for software development process and their associated principles and techniques have provided much better understanding of the activities in the process. All the concepts and principles highlighted above are going to be applied while developing software using object-oriented approach. basic understanding of the generic concepts and principles that apply object-oriented approach.

Activity Assessment

i. Define the following terms:

(a) Polymorphism

(b) Modularity

ii. Describe the structure of any agency or organization as a hierarchy. Include at least three levels.

iii. What are the four basic principles of object orientation? Provide a brief description of each

iv. What are the four basic concepts of object orientation? Provide a brief description of each

v. Which of the following statements are true? For those that are false, explain why.

(a) Trainee is an example of a class.

(b) DoraCheong is an example of a class.

(c) "createJob" is an example of an object.

(d) "deleteFile" is an example of an operation

(e) "name" is an attribute

vi. Suppose we wish to model an application for issuing business registration licenses. Identify:

(a) Three classes for the model

(b) At least three attributes for each class

Activity 4 – Associations and Links

Introduction

A class by itself is not very useful. A large software system may have thousands of classes. Objects in different classes must relate to each other, interact and collaborate to carry out processes. Relationships between object classes (known as associations) are shown as lines linking objects. Association expresses relationships between classes and defines links instances of classes (objects). Link expresses relationships between objects. There are four kinds of relations between classes:

a. Association

b. Aggregation

c. Composition

d. Generalization

Association

This is the simplest form of relation between classes. It is a peer-to-peer relationship. It is a structural relationship describing a set of links. Links are connections between objects.

One object is aware of the existence of another object. It is being implemented in objects as references. Association is a relationship between two or more classifiers that involves connection among instances. Associations between classes A and B can be (Ojo and Estevez, 2005):

- A is a physical or logical part of B
- A is a kind of B
- A is contained in B
- A is a description of B
- A is a member of B
- A is an organization subunit of B
- A uses or manages B
- A communicates with B
- A follows B
- A is owned by B

Figure 1.12 shows an example of association where a Person works for the Company

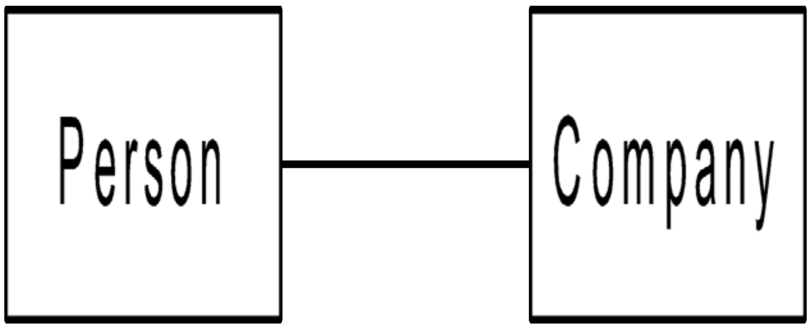


Figure 1.12: Example of a Simple Association

Aggregation

Aggregation is a restrictive form of "whole-part" or "part-of" association. Objects are assembled to create a more complex object. Assembly may be physical or logical. Aggregation defines a single point of control for participating objects. The aggregate object coordinates its parts. Aggregation is represented as a hollow diamond, pointing at the class which is used. Figure 1.13 shows examples of aggregation relationship.

Example:

- A CPU is part of a computer.

- CPU, devices, monitor and keyboard are assembled to create a computer
- An engine is a part of a car

Composition

Composition is a stricter form of aggregation. The lifespan of individual objects depend on the lifespan of the aggregate object. A part cannot exist on its own. A composite class is built of other classes. The class needs one or more other classes to exist. Composition is represented as a solid diamond. Figure 1.14 shows examples of composition relationship.

Example:

- A word cannot exist if it is not part of a line.
- If a paragraph is removed, all lines of the paragraph are removed, and all words belonging to that line are removed.

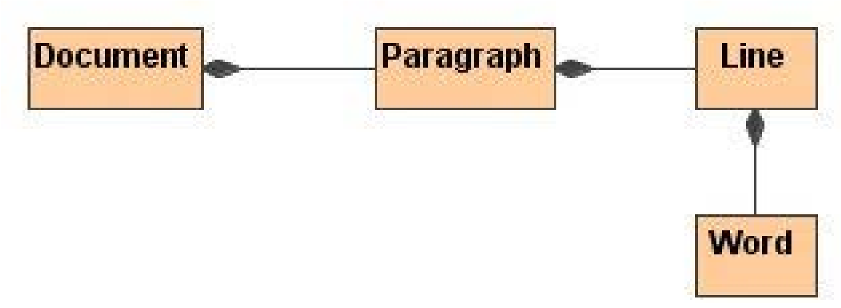


Figure 1.14: Example of Composition Relationship (Source: Ojo and Estevez, 2005)

Inheritance or Generalization

Generalization is also called "inheritance" relationship. A generalization is the relationship between a general class and one or more specialized classes. In a generalization relationship, the specializations are known as subclass and the generalized class is known as the superclass. A generalization allows the inheritance of the attributes and operations of a superclass by its subclasses. It is equivalent to a "kind-of" or "type-of" relationship

Inheritance or generalization is described as a parent/child relationship, where a child class inherits all the data members and methods of a class, and adds its own to create new behaviour. Inheritance is represented as a hollow triangular arrow, with the point attached to the parent class. Figure 1.15 shows an example of a inheritance relationship

Example

- Common features are defined in User.
- FrontOfficeEmployee and BackOfficeEmployee inherit them.

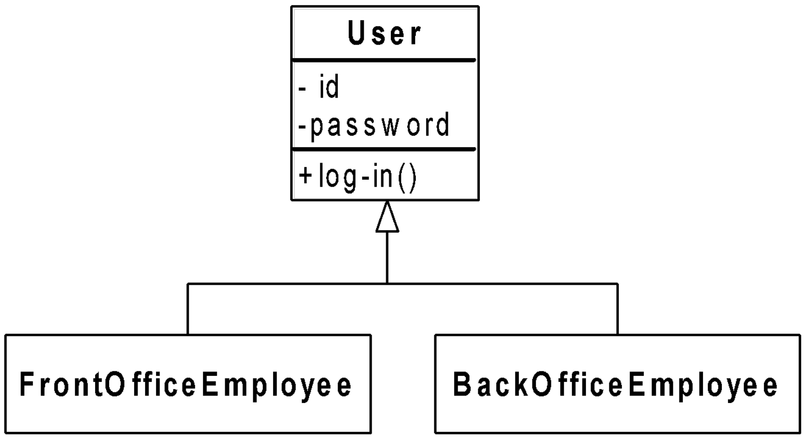


Figure 1.15: Example of Generalization Relationship (Source: Ojo and Estevez, 2005)

Super Class Versus Sub Class

Super Class is a class that contains the features common to two or more classes. A super-class is similar to a superset, e.g. agency-staff. Sub-Class is a class that contains at least the features of its super-class(es). A class may be a sub-class and a super-class at the same time.

Multiplicity

Multiplicity shows how many objects of one class can be associated with one object of another class. Example as shown in Figure 1.17: A citizen can apply for one or more licenses, and a license is required by one citizen.

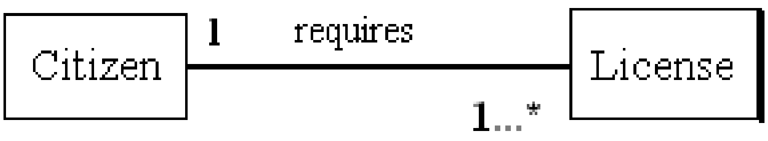
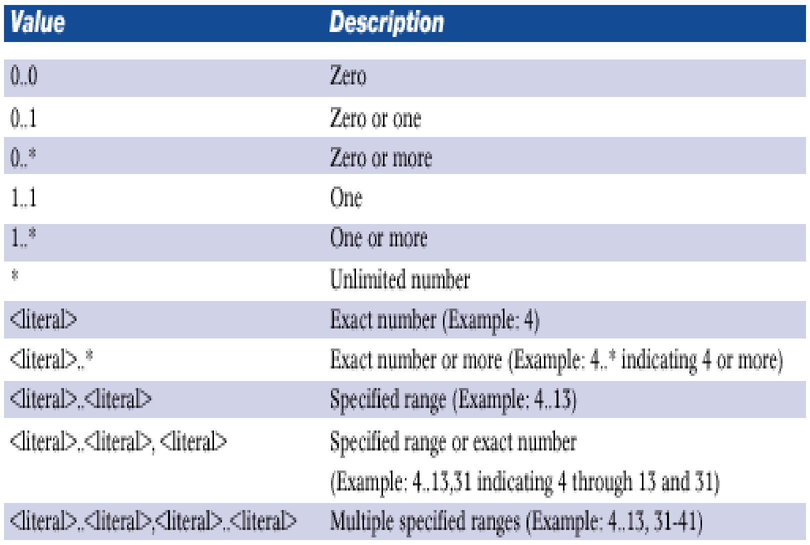


Figure 1.17: Example of the Use of MultiplicityExample of multiplicity is as shown in Figure 1.18:



Value	Description
0..0	Zero
0..1	Zero or one
0..*	Zero or more
1..1	One
1..*	One or more
*	Unlimited number
<literal>	Exact number (Example: 4)
<literal>..*	Exact number or more (Example: 4..* indicating 4 or more)
<literal>..<literal>	Specified range (Example: 4..13)
<literal>..<literal>, <literal>	Specified range or exact number (Example: 4..13,31 indicating 4 through 13 and 31)
<literal>..<literal>,<literal>..<literal>	Multiple specified ranges (Example: 4..13, 31-41)

Figure 1.18: Multiplicity Values and Description

Conclusion

As stated by Liu,(2001), the purpose of an association and a link between two objects may have many sorts of relationship, and thus classes (or concepts) may have all sorts of associations in a problem domain. Whether an association is useful or not depends on whether it fulfills the following purpose:

- An association between two classes is to provide physical or conceptual connections between objects of the classes.
- Only objects that are associated with each other can collaborate with each other through the links.

Booch describes the role of a link between objects as follows:

"A link denotes the specific association through which one object (the client) applies the services of another object (the supplier), or through which one object may navigate to another".

Activity Assessment

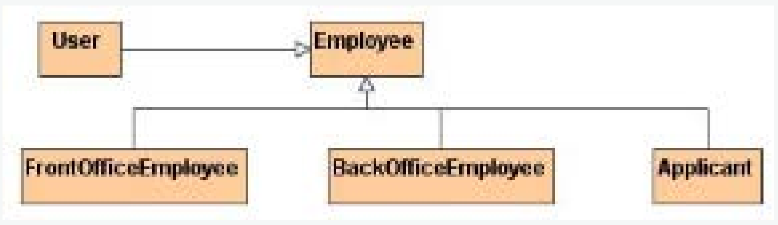
i. Identify which of the following statements are true. For those that are false, explain why.

(a) There is an association between Trainee and Course

(b) There is a composition between Course and Professor

(c) There is an aggregation between Course and Venue

ii. Suppose we wish to model an e-service application for a government agency. Does this diagram reasonably model the relationship between the entities User, Employee, FrontOffice, Employee, Back-Office Employee and Applicant? If not, provide a more appropriate model



iii. Consider the generalization hierarchy provided in assessment 1.4.5 above.

(a) Introduce an operation for the super-class which could be implemented in different ways by its sub-classes.

(b) Provide an example of polymorphism. Explain.

iv. List and describe four basic UML relationships

UNIT SUMMARY

This unit has presented principles of Object Orientation: abstraction, encapsulation, modularity and hierarchy and fundamental concepts of Object Orientation: objects, classes, inheritance and Polymorphism. Object is any abstraction that models a single thing in a universe with some properties and behaviour. A class is any uniquely identified abstraction of a set of logically related objects that share similar characteristics. Classes may be related by the following types of relationships: association, aggregation, composite and inheritance or generalization.

Unit Assessment

Instructions

Answer the following questions.

i. Describe an overview of the project you are going to use while practicing object-oriented in software development (Answer: section 1.1.1)

ii. With an example differentiate between the following terms:

(a) Structured approach and object-oriented approach (Answer: section 1.2.4 and 1.2.5)

(b) Aggregation and composition

(Answer: sections 1.4.3 and 1.4.4)

iii. Clearly explain the modularity term in object-oriented approach

(Answer: section 1.3.2.3)

iv. Define the following terms. Provide an example in each.

(a) Abstraction (Answer: section 1.3.2.1)

(b) Polymorphism (Answer: section 1.3.3.4)

(c) Encapsulation (Answer: section 1.3.2.2)

(d) Inheritance (Answer: sections 1.3.3.3 and 1.4.5)

v. Define the term "multiplicity" and list two commonly used multiplicity values.

(Answer: section 1.4.7)

Answers

i.Describe an overview of the project you are going to use while practicing object-oriented in software development Overview will differ from one project to another project from students' case projects

ii. With an example differentiate between the following terms:

(a) Structured approach and object-oriented approach (4 marks)

Structured Approach

- The approach is analyzing and designing software systems through functional decomposition – i.e. examining an information system in terms of the functions it performs and the data it uses and maintains.
- Analyst identifies the major functions or processes of a system, then breaks or decomposes each function down into its smaller composite steps
- Object-oriented (OO) approach
- The approach decomposes the system down into a set of objects – i.e. it examines the system in terms of the things in the system and how these things act and interrelate with each other to provide some higher level behaviour. In object-oriented approach:
- The analyst first identifies the objects that comprise the system, then They create an object model which groups the objects into classes, and describes each class in terms of its attributes (or data), methods (or functions), and relationships to other classes.

Aggregation and composition

Aggregation

Aggregation is a restrictive form of "whole-part" or "part-of" association. Objects are assembled to create a more complex object. Assembly may be physical or logical. Aggregation defines a single point of control for participating objects. E.g. a CPU is part of a computer.

Composition

Composition is a stricter form of aggregation. The lifespan of individual objects depend on the lifespan of the aggregate object. A part cannot exist on its own. E.G. Paragraph cannot exist if there a no lines of words

iii. Clearly explain the modularity term in object-oriented approach (3 marks)

Modularity deals with the process of breaking up complex systems into small, self contained pieces that can be managed easily. Modularity can be done by:

- Decomposing the problem into smaller sub-problems
- Trying to solve each sub-problem separately Each solution is a separate component that includes
- Interface: types and operations visible to the outside
- Specification: intended behavior and property of interface
- Implementation: data structures and functions hidden from outside

iv. Define the following terms. Provide an example in each.

(a) Abstraction (2 marks)

Abstraction is a model that includes most important aspects of a given system while ignoring less important details. The skill of abstraction is important to capture the essences and ignore the non-essences. It is the process of allowing to focus on most important aspects while ignoring less important details. It allows management of complexity by concentrating on essential characteristics that makes an entity different from others.

Example can be a mobile phone whereby users are just seeing an interface

(b) Polymorphism (2 marks)

Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data used in operation.

For example: Consider the operation of addition.

- For two numbers, the operation will generate a sum
- If the operands are strings, then the operation would produce a third string by concatenation

(c) Encapsulation (2 marks)

In encapsulation, only the instance that owns an item of data is allowed to modify or read it. Encapsulation means that all of this information is packaged under one name and can be reused as one specification or program component. In other words, encapsulation separates implementation from users or clients

Example:

The object chair (and all objects in general) encapsulates data (the attribute values that define the chair), operations (the actions that are applied to change the attributes of chair), other objects (composite objects can be defined), constants (set values), and other related information.

(d) Inheritance (2 marks)

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification.

For example:

The grandchild is a part of the class parent which is again a part of the class grandparent.

v. Define the term “multiplicity” and list two commonly used multiplicity values. (2 marks)

Multiplicity shows how many objects of one class can be associated with one object of another class.

Examples are:

- 0…1 – zero to one
- 1…* - One or more

Unit Readings and Other Resources

- Liu Z., (2001), “Object-Otiented Software Development Using UML”, The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.
- Ojo A. and Estevez E., (2005), “Object-Oriented Analysis and Design with UML”, Training Course, The United Nations University, UNU-IIST International Institute for Software Technology, e-Macao Report 19, Version 1.0, October.
- Ariadne Training (2001), “UML Applied Object Oriented Analysis and Design Using the UML”, Ariadne Training Limited

Unit II: Fundamentals of UML

Unit Introduction

The Unified Modeling Language (UML) has become the universally-accepted language for software design blueprints. It has emerged as the standard notation for modeling. UML is the visual language used to convey design ideas, which emphasizes how developers really apply frequently used UML elements, in software development process.

Unit Objectives

Upon completion of this unit you should be able to:

1. Define unified Modelling Language
2. Describe the purpose of UML
3. Differentiate between static and dynamic UML diagrams
4. Categorize UML diagrams into SDLC phases

KEY TERMS

UML

Unified Modelling Language (UML) is a graphical modelling language that provides syntax for describing the major elements (called artifacts in UML) of software systems.

Modelling

Modelling is the designing of software applications before coding (implementation in a particular programming language). A model is a representation or simplification of reality.

Diagram

A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). You draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system.

Activity 1 – UML as a Modelling Tool

Introduction

The norm for engineering field is to model first before implementation. Modeling can be mathematically or graphically, This unit presents a Unified Modeling Language as a language to model object-oriented artifacts.

Engineering Norm

For many years, the term object oriented (OO) was used to denote a software development approach that used one of a number of object-oriented programming languages (e.g., Java, C++). Today, the OO paradigm encompasses a complete view of software engineering.

The norm for engineering analysis and design systems emphasizes modelling of the system first before implementation. Modelling is a proven and well-accepted engineering technique. Modelling can be done mathematically or by any other common notation understood by many engineers of the same field worldwide.

Software Engineering had lacked such a notation. Between 1989 and 1994, more than 50 software modelling languages were in common use – each of them carrying their own notations. Each language contained syntax peculiar to itself, whilst at the same time, each language had elements which bore striking similarities to the other languages, and bad enough neither of these languages were complete (Ariadne Training, 2001).

UML Background

In the mid 1990's, three methods emerged as the strongest. Ariadne Training (2001) state that these three methods had begun to converge, with each containing elements of the other two. Each method had its own particular strengths:

- Booch was excellent for design and implementation. Grady Booch had been a major player in the development of Object Oriented techniques for the language.
- Object Modelling Technique (OMT) was best for analysis and data-intensive information systems.
- Object Oriented Software Engineering (OOSE) featured a model known as Use Cases. Use Cases are a powerful technique for understanding the behavior of an entire system (an area where OO has traditionally been weak).

So, the Unified Modelling Language (UML) is largely the product of three well known software engineers, - Grady Booch, Ivar Jacobson and James Rumbaugh. In 1994, James Rumbaugh, the creator of OMT joined Grady Booch at Rational Corp. The aim of the partnership was to merge their ideas into a single, unified method. By 1995, the creator of OOSE, Ivar Jacobson, had also joined Rational, and his ideas (particularly the concept of "Use Cases") were fed into the new Unified Method – now called the Unified Modeling Language (Ariadne Training, 2001).

The Unified Modelling Language or the UML as a graphical modelling language aimed at providing syntax for describing the major elements of software systems (called artifacts in the UML). The UML represents a collection of best engineering practices that have proven successful in the modelling of large and complex systems. In this course we need to explore the main aspects of the UML, and describe how the UML can be applied to software development process.

What is UML?

Booch et al, (2005) defines the Unified Modelling Language (UML) to be a language for: Specifying, Visualizing, Constructing and Documenting the artifacts of a software-intensive system. It is a standard language for writing software blueprints. It is a graphical language for capturing the artifacts of software development. UML is the de-facto standard for Object Oriented modelling.

a) The UML as a Language for Visualizing:

For many programmers, the distance between thinking of an implementation and then pounding it out in code is close to zero. You think it, you code it. In fact, some things are best cast directly in code. Text is a wonderfully minimal and direct way to write expressions and algorithms

b) The UML as a Language for Specifying:

Specifying means of building models that are precise, unambiguous, and complete. In particular, the UML addresses the specification of all the important analysis, design, and implementation decisions that must be made in developing and deploying a software-intensive system.

c) The UML as a Language for Constructing:

The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages. This means that it is possible to map from a model in the UML to a programming language such as Java, C++, Visual Basic or PHP, or even to tables in a relational database or the persistent store of an object-oriented database.

Things that are best expressed graphically are done so graphically in the UML, whereas things that are best expressed textually are done so in the programming language. This mapping permits forward engineering: The generation of code from a UML model into a programming language.

d) The UML as a Language for Documenting:

A healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifact include (but are not limited to): Requirements, Architecture, Design, Source code, Project plans, Tests, Prototypes, Releases.

Depending on the development culture, some of these artifacts are treated more or less formally than others. Such artifacts are not only the deliverables of a project, they are also critical in controlling, measuring, and communicating about a system during its development and after its deployment.

The UML addresses the documentation of a system's architecture and all of its details. The UML also provides a language for expressing requirements and for tests. Finally, the UML provides a language for modelling the activities of project planning and release management.

The UML can also be a Language for Communication. That is communication with customers has proven to be equally problematic. Graphical modelling can make people (technical and non-technical) understand the artifacts of the software system expected. In software development, some of the things that require communication include requirements, design, implementation, and deployment. UML is a language designed to communicate these things.

As with any language, the UML has its own notation and syntax. It does not tell you how to develop software. It can be applied in any software development processes; waterfall model, spiral model, iterative, incremental frameworks. Its notation comprises a set of specialized shapes for constructing different kinds of software diagrams. Each shape has a particular meaning. UML is a generic, broad language enabling the key aspects of a software development to be captured on paper

Goals of UML by Ojo and Estevez, (2005) are to:

- Provide modelers with an expressive, visual modelling language to develop and exchange meaningful models
- Provide extensibility and specialization mechanisms to extend core concepts
- Support specifications that are independent of particular programming languages and development processes
- Provide a basis for understanding specification languages
- Encourage the growth of the object tools market
- Supports higher level of development with concepts such as components frameworks or patterns

UML with Modelling

Modelling is the designing of software applications before coding (implementation in a particular programming language). A model is a representation or simplification of reality. It provides a blueprint of a system. A model does not dictate or show how the implementation will actually be done. It just shows what, who, which, where, when etc. Model-driven analysis and design emphasizes the drawing of pictorial system models to document and validate both existing and/or proposed systems. Ultimately, the system model becomes the blueprint for designing and constructing an improved system (Booch et al, 2005).

Modelling is a central part of all the activities that lead up to the deployment of good software. We build models to (Booch et al, 2005):

- Communicate the desired structure and behavior of our system.
- Visualize and control the system's architecture.
- Better understand the system we are building, often exposing opportunities for simplification and reuse.
- Manage risk.
- Modelling manages Complexity
- Modelling Promotes Reuse

Modelling ensures that:

- Business functionality is complete and correct,
- End-user needs are met, and
- Program design meets requirements for scalability, robustness, security, extensibility, and other characteristics, all these must be ensured before implementation in code

There are many elements that contribute to a successful software organization; one common thread is the use of modeling. Modeling is a proven and well-accepted engineering technique. There are three basic building blocks: Elements which are main "citizens" of the model, relationships i.e. relationships that tie elements together and Diagrams which are mechanisms to group interesting collections of elements and relationships. These elements are used to represent complex structures.

Principles of Modelling

The use of modeling has a rich history in all the engineering disciplines. That experience suggests four basic principles of modeling (Booch et al, 2005):.

(a) The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped. In other words, choose your models well. The right models will brilliantly illuminate the most wicked development problems, offering insight that you simply could not gain otherwise; the wrong models will mislead you, causing you to focus on irrelevant issues.

(b) Every model may be expressed at different levels of precision.

The best kinds of models are those that let you choose your degree of detail, depending on who is doing the viewing and why they need to view it. An analyst or an end user will want to focus on issues of what; a developer will want to focus on issues of how. Both of these stakeholders will want to visualize a system at different levels of detail at different times.

(c) The best models are connected to reality.

In structured analysis techniques there is a basic disconnect between its analysis model and the system's design model. Failing to bridge this break causes the system as conceived and the system as built to diverge over time. In object oriented systems, it is possible to connect all the nearly independent views of a system into one semantic whole.

(d) No single model is sufficient.

Every nontrivial system is best approached through a small set of nearly independent models. To understand the architecture of object-oriented software system, you need several complementary and interlocking views: a use case view (exposing the requirements of the system), a design view (capturing the vocabulary of the problem space and the solution space), a process view (modeling the distribution of the system's processes and threads), an implementation view (addressing the physical realization of the system), and a deployment view (focusing on system engineering issues). Each of these views may have structural, as well as behavioral, aspects. Together, these views represent the blueprints of software.

Conclusion

The purpose of the Unified Modeling Language is to visualize, specify, construct, document and communicate object-oriented systems and it is gaining adoption as a standard language. The language provides the notations to produce models.

Activity Assessment

i. Explain why UML is being used in software engineering

ii. List three main software engineers who made contributions to the creation of UML and describe their potential contributions

iii. Explain why modelling is being emphasized in software engineering

iv. Write three goals of a UML

Activity 2 - UML Diagrams

Introduction

A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). You draw diagrams to visualize a system from different perspectives, so a diagram is a projection into a system. UML has a lot of different diagrams (models). The reason for this is that it is possible to look at a system from different viewpoints. UML being a graphical language includes nine such diagrams (models):

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

UML nine diagrams can be divided into two categories

(a) Four diagram types represent static application structure:

- Class Diagram
- Object Diagram
- Component Diagram
- Deployment Diagram

(b) Five represent different aspects of dynamic behaviour

- Use Case Diagram
- Sequence Diagram
- Activity Diagram
- Collaboration Diagram
- Statechart Diagram

A Class Diagram

In Object Oriented design and development terms, a class has a name, a set of methods (also known as operations) and related data members (also known as attributes) as shown in Figure 2.1. Class by itself is not very useful. A large software system may have thousands of classes, Modelling the relationships (association, inheritance, composition or aggregation) between them really defines systems behaviour.

Class diagrams shows a set of classes, interfaces, and collaborations and their relationships. Class diagrams are the most common diagrams found in modelling object-oriented systems. It is an essential aspect of any Object Oriented Design method.

Class diagrams address the static design view of a system. They are used at the analysis stage as well as design. Class diagrams that include active classes address the static process view of a system. Class Diagram syntax are being used to draw a plan of the major concepts for anyone to understand. This is called the Conceptual Model. Together with use cases, a conceptual diagram is a powerful technique in requirements analysis. Figure 2.2 shows an example of a class diagram

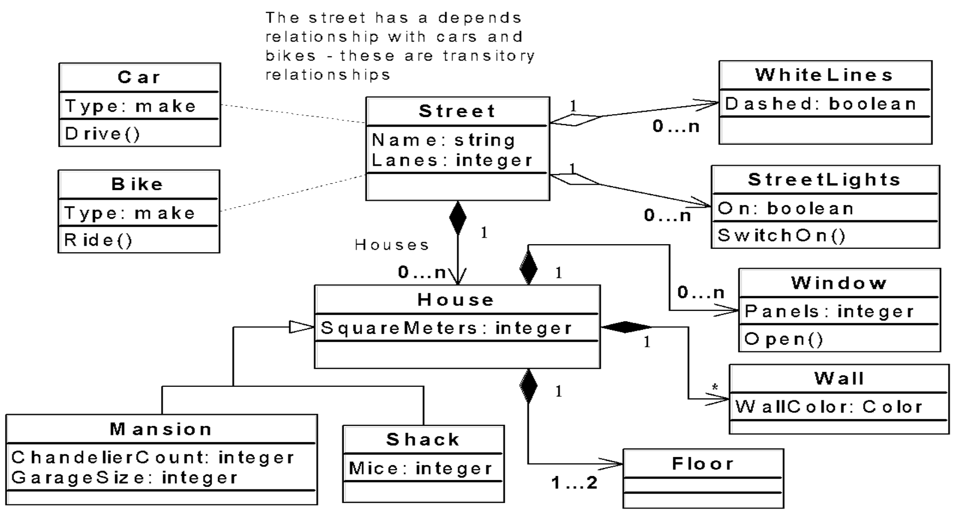


Figure 2.2: An Example of a Class Diagram2.2.3

Object Diagram

Object Diagrams show a set of objects and their relationships. They are static snapshots of element instances found in class diagrams. Figure 2.3 shows an example of an object diagram.

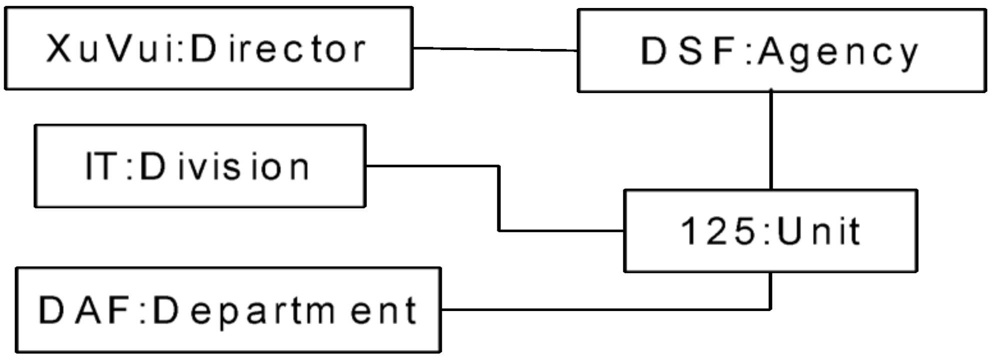


Figure 2.3: Example of a UML Object Diagram

A Use Case Diagram

Use cases are versatile and valuable techniques for describing user requirements. A use case is a high-level description of a major user requirement. It represents the functionality of the system. It is a description of the system's behaviour from a user's viewpoint and constitutes a complete interaction with the system initiated by a user or another system.

Use case diagrams address the static use case view of a system. The different types of people and/or devices (called actors) that interact with the system are identified along with the functions that they perform or initiate. A Use Case diagram shows a set of use cases and actors (a special kind of class) and their relationships.

These diagrams are especially important in organizing and modelling the behaviors of a system. They are valuable aid during analysis, since developing Use Cases helps to understand requirements. Use Cases and a Conceptual Model are the powerful techniques in requirement analysis. Notations used when representing use case diagrams include:

Basic Use Case Notation

The Actor represents a user of the system, or any external system that interacts with the system. The Usecase represents a piece of functionality that is important to the user. Mostly we see the actor as a human user, but it can also represent a system or other nonhuman artifact. Figure 2.4 shows the basic notation of a use case diagram.

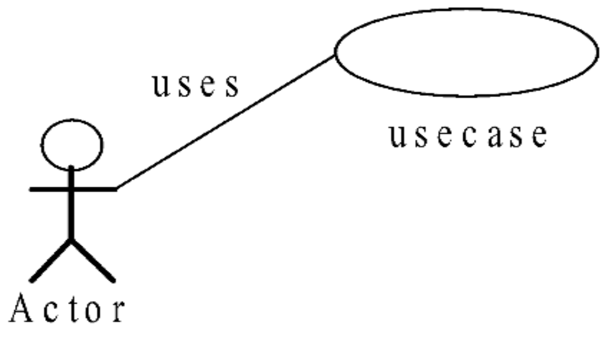


Figure 2.4: Basic Notation of a Use Case Diagram

Using the <<uses>> Relationship

This feature encourages re-use. If a use-case needs the functionality of another use-case in order to perform its task, it "uses" the 2nd use case. The relationship is drawn as a line with arrowhead pointing to the use case that is being "used" as shown in Figure 2.5.

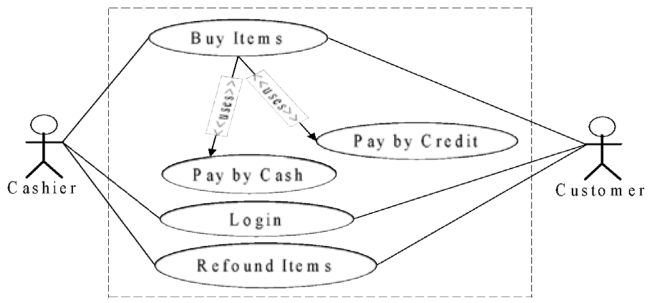


Figure 2.5: A <<uses>> Relationship in a Use Case Diagram

Using the <<extends>> relationship

The "extends" notation extends the functionality of a use case to deal with errors or exceptions. "extends" relationship as shown in Figure 2.6 is being used when there is one use case that is similar to another but does a bit more. The relationship is drawn as a line with arrowhead pointing to the major use case.

The following is the essence of the "extends" relationship:

- Capture the normal simple case first
- For every step in that use case, ask "What could go wrong here" and "How might this work out differently?"
- Plot all variations as extensions of the given use case.

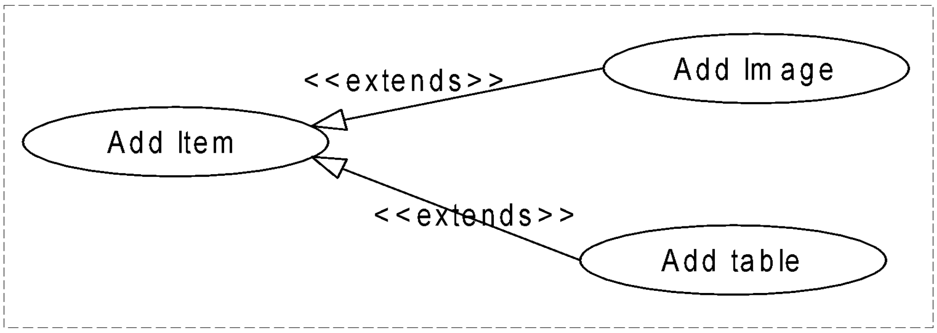


Figure 2.6: An <<extends>> Relationship in a Use Case Diagram

Using the << include >> relationship

"include" relationship as shown in Figure 2.7 can be used for making up a big use case from simpler ones.

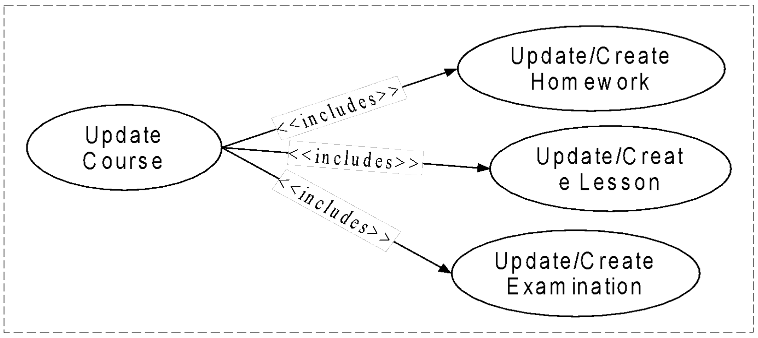


Figure 2.7: An <<includes>> Relationship in a Use Case Diagram

Use Case diagrams serve a number of purposes. Use case diagrams provide:

- A graphical overview of system functionality.
- A bridge between developers and client/user. The notation is sufficiently simple to enable a non-user to understand and comment.
- A starting point for developing detailed requirements.
- A basis for developing design.
- Integration test cases.

Use Case Scenario

A scenario is a description of a use case as a story. It is a story of one user's interaction with the system. It can be described in terms of:

- The actors involved, and
- The steps to be taken in order to achieve the functionality described in the use-case title. (Use cases are about functionality, so make sure the title reflects some functionality.)

The steps involved usually take the form of the normal flow of events, followed by alternate flows and/or exception flows.

Example:

In an ATM banking system, a use case would be "Validate User". The steps involved in authenticating a user are described in scenarios. There will be a number of different scenarios for "Validate User", describing different situations that can arise.

Main (primary) flow of events for "validate user" use case could be:

1. System prompts user for PIN number
2. User enters PIN number via keypad.
3. User commits PIN by pressing the Enter key.
4. System then checks PIN to see if it is valid.
5. If the PIN is valid, system acknowledges the entry.
6. End of Use Case

Exception Flow of events

1. System prompts user for PIN number
2. User enters PIN number via keypad.
3. User presses Cancel button to cancel transaction
4. End of Use case. No changes made to user account.

By using scenarios, you are able to discover the objects that must collaborate to produce the results needed.

Sequence and Collaboration Diagrams

When developing Object-Oriented software, anything our software needs to do is going to be achieved by objects collaborating. We can draw a collaboration diagram to describe how we want the objects we built to collaborate.

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

- They both describe the flow of messages between objects.
- They are very useful for describing the procedural flow through many objects.
- Interaction diagrams consisting of a set of objects and their relationships, including the messages that may be dispatched among them
- Both are interaction diagrams which address the dynamic view of a system
- They are models that describes how a group of objects collaborate in some behaviour, typically a single use case

Sequence Diagrams

A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages. Sequence diagrams focus on the order in which the messages are sent. They provide a sequential map of message passing between objects over time. The Sequence Diagrams are driven by the Use Cases which are the system requirements. In this form objects are shown as vertical lines with the messages as horizontal lines between them. The sequence of messages is indicated by reading down the page (read left to right and descending). Sequence Diagrams are about deciding and modelling "how" the system will achieve "what" we described in the Use Case model.

Although there is no fixed recipe for developing Sequence Diagrams, we can follow an approach that will result in a logical sequence diagram. This is as follows:

- Take the Use Case description and turn it into simple pseudo code running down the right hand side of the State Diagram.
- Guess which classes you think might be involved - based on the content of the Use Case description. Simple noun analysis is as good a way to start as any.
- For each of the steps in you pseudo code decides which of the classes should have the responsibility for doing that task.
- For each of those tasks you may want to go back and decide to break them down into a number of simpler tasks.
- Add in probes that correspond to the "uses (includes)" and "extends" relationships in the Use Case diagram.
- Consider any important errors you might have to handle that perhaps weren't covered in the Use Case model.
- Consider whether anything you have discovered needs to be fed-backed into the Use Case model.

An analysis of the Sequence Diagrams shows the following (Showmn in Figure 2.8):

i. A Class (Figure 2.8(a))

- Participates in a sequence by sending and/or receiving messages
- Is placed at the top of the diagram and is shown using a rectangle with a descriptive name.

ii. A Lifeline (Figure 2.8(b))

- Denotes the life of an object during a sequence
- Is a dotted vertical line below each class.

iii. A focus of control (Figure 2.8(c))

- Is a long, narrow rectangle placed atop a lifeline
- Denotes when an object is sending or receiving messages

iv. A message (Figure 2.8(d))

- Conveys information from one object to another object
- Is depicted using a horizontal arrow labeled with the message description and

  applicable parameters.

Example of a "Make a Cup of Tea" sequence diagram generated from its corresponding use case description is as shown in Figure 2.9.

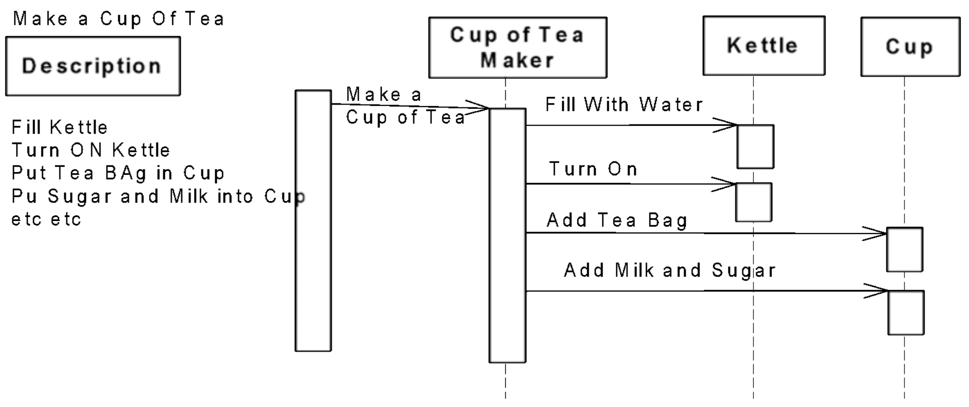


Figure 2.9: A Sequence Diagram for "Make a Cup of Tea" Use Case

Collaboration Diagram

A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Collaboration diagrams express both the context of a group of objects and the interaction between these objects. It focuses upon the relationships between the objects. They are very useful for visualizing the way several objects collaborate to get a job done and for comparing a dynamic model with a static model. When creating collaboration diagrams, patterns are used to justify relationships. Patterns are best principles for assigning responsibilities to objects. Figure 2.10 shows an example of a collaboration diagram.

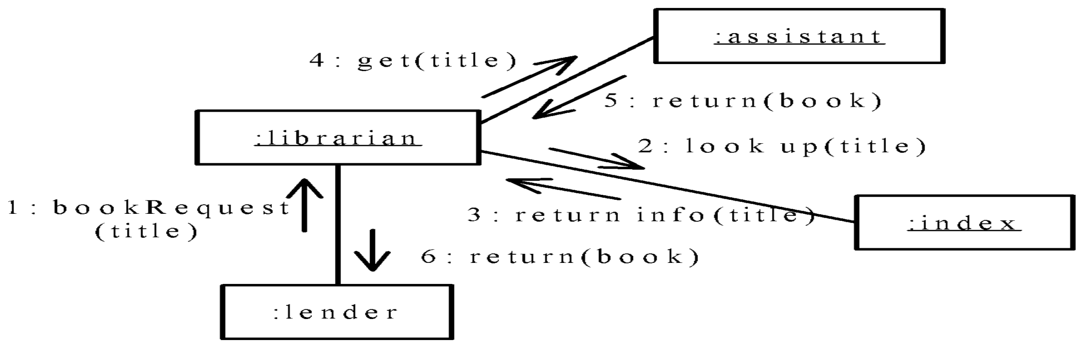


Figure 2.10: Example of a Collaboration Diagram

Note for Sequence and Collaboration Diagrams

- Interaction diagrams require use cases, system operation contracts, and domain (class or conceptual) model to already exist
- Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.
- Collaboration and sequence diagrams describe the same information, and can be transformed into one another without difficulty.
- The choice between the two depends upon what the designer wants to make visually apparent.

Statechart Diagrams

Statechart is a diagram that shows all possible object states. Some objects can at any particular time be in a certain state. A Statechart or simply a state diagram shows a state machine, consisting of states, transitions, events, and activities. Statechart diagrams address the dynamic view of a system. UML State charts are not normally needed. They are needed when an object has a different reaction dependent on its state an example of a Statechart diagram is shown in Figure 2.11.

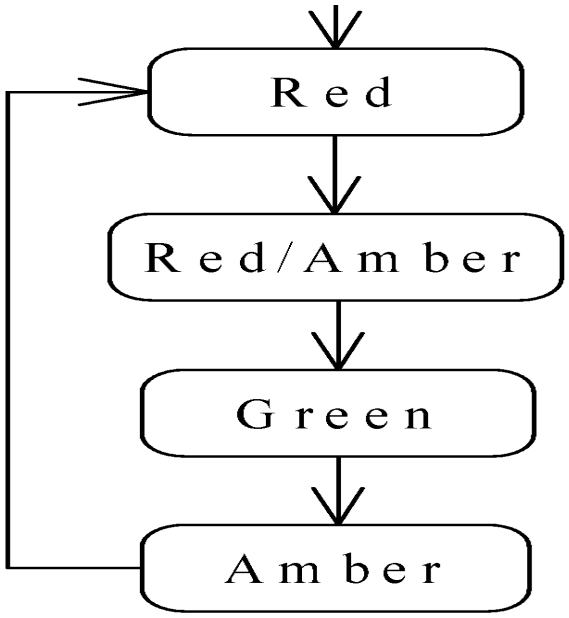


Figure 2.11: Example of a Statechart Diagram **Activity Diagrams**

An activity diagram is a special kind of a statechart diagram that shows the flow from activity to activity within a system. They are used to show how different work flows or processes in a system are constructed, how they start, the many decision paths that can be taken from start to finish and where parallel processing may occur during execution.

Activity diagrams address the dynamic view of a system. They are especially important in modelling the function of a system and emphasize the flow of control among objects. An Activity diagram as shown in Figure 2.12 generally does not model the exact internal behaviour of a software system (like a Sequence diagram does) but rather it shows the general processes and pathways at a high level

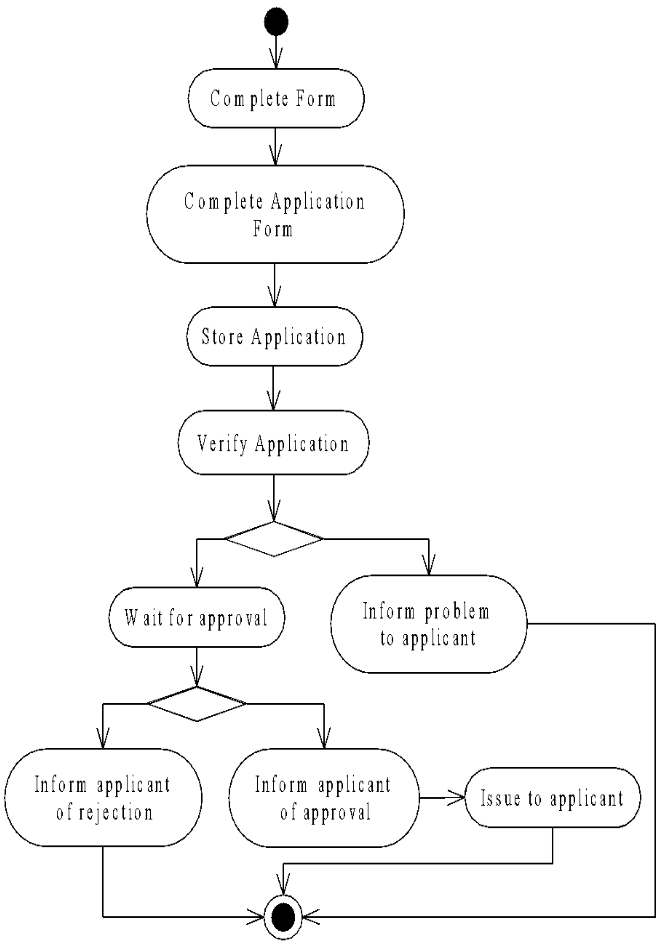

Complete Form
Complete Application Form
Store Application
Verify Application
Wait for approval
Inform problem to applicant
Inform applicant of rejection
Inform applicant of approval
Issue to applicant

Figure 2.12: Example of Activity Diagram

Package Diagrams

Any non-trial system need to be divided up in smaller, easier to understand "chunks". A package is basically a logical container into which related elements can be placed, "like a folder or directory in an operating system". We can display groups of packages and relationships between then on the UML package diagram.

Packages does not show actually what is inside the package, it provides a very "high-level" view of the system. Some case tools allow the user to double-click on the package icon in order to open-up the package and explore the contents. The common use of a package is to group related classes together, sometimes group related use cases.

Packages can be used to:

- Group large systems into easier to manage subsystems
- Allow parallel iterative development

Package diagram in Figure 2.13 shows three UML packages representing a "three-tier model"

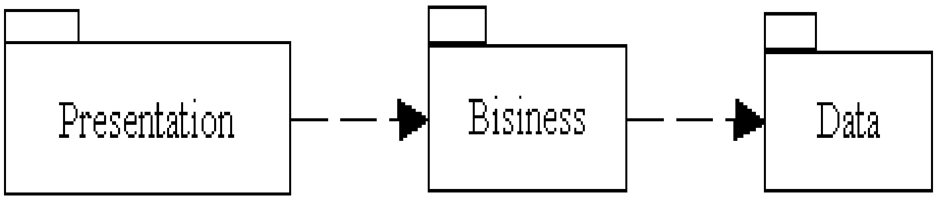


Figure 2.13: Example of a Package Diagram

Component Diagrams

A component diagram is similar to the package diagram. It works in the same way as the package diagram, showing the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. Component diagrams emphasize the physical software entity e.g. files headers, executables, link-libraries etc, rather than the logical partitioning of the package diagram. It is based heavily on the package diagram, but has added ".dll" to handle I/O, and has added a test harness executable. Not heavily used, but can be helpful in mapping the physical, real life software code and dependencies between them. Figure 2.14 shows a symbol used for a software component.

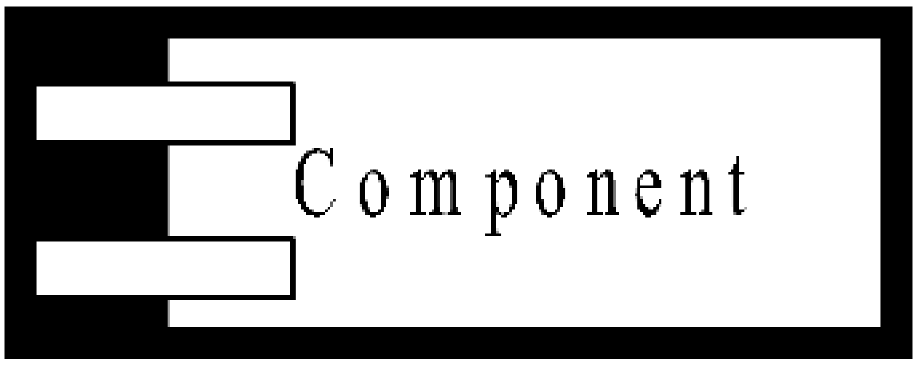


Figure 2.14: A Symbol for a Software Component

Deployment Diagrams

Deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of an architecture. They are related to component diagrams in that a node typically encloses one or more components. Figure 2.15 shows a node symbol used in deployment diagram.

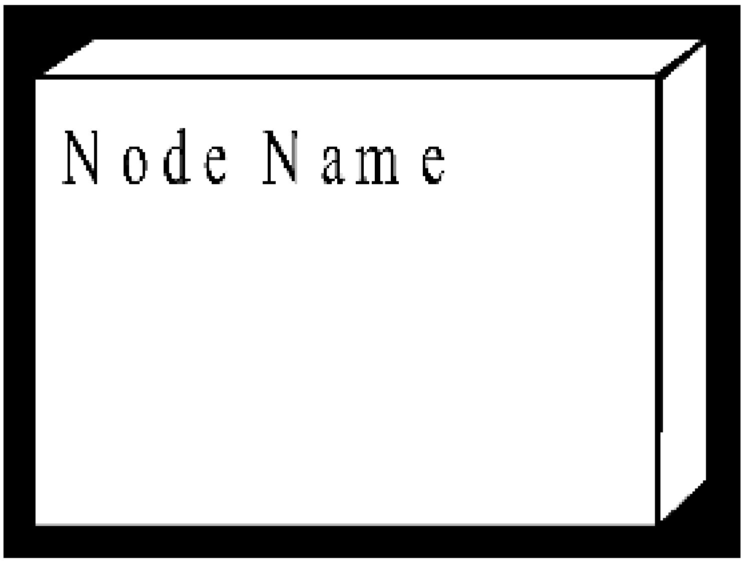


Figure 2.15: A Node Symbol for a Deployment Diagram

Design Class Diagrams

We built a class (conceptual or Domain) diagram made using concepts having properties of these concepts. No behaviour was allocated to any of these concepts. After creating collaboration diagrams, we can progress the conceptual model, and build it into a true "Design Class Diagram (DCD)". DCDs are based on the collaboration diagram. Attribute visibility is shown for permanent connections. DCD is a diagram which we can base our final program code upon. We can say that a DCD is a modification of a class diagram

Summary on UML Diagrams

In this activity, you have learnt about UML diagrams and their use context. Each diagram in UML provides a certain view of the software under development. When creating a diagram, the following can be a guide question for each:

- Use Cases – How will our system interact with the outside world?
- Class Diagram – What objects do we need? How will they be related?
- Collaboration Diagram – How will the objects interact?
- Sequence Diagram – How will the objects interact?
- Statechart (or state) Diagram – What states should our objects be in?
- Component Diagram – How will our software components be related?
- Deployment Diagram – How will the software be deployed

Activity Assessment

i. Consider a software process consisting of the following activities: requirement gathering, object oriented analysis, object design, implementation and deployment.

(a) List the diagrams that are essential for each of these activities.

(b) Provide justifications for your choice of diagrams.

ii. Select the best answer:

1. The class diagram:

(a) is the first model created in the project

(b) is created after the other models

(c) is used to specify objects and generate code

(d) is used to create the sequence and the collaboration diagrams.

2. The sequence diagram models:

(a) the sequence of activities to implement the model

(b) the way that objects communicate

(c) the relationships among objects

(d) the order in which the class diagram is constructed

3. The collaboration diagram:

(a) is a unique view of object behaviour

(b) models the connections between different views

(c) models the relationships between software and hardware components

(d) models the way objects communicate

iii. What is the role of State transition and interaction diagrams in system development?

UNIT SUMMARY

The UML is a graphical language to support modeling of software documents in object-oriented approach. The language provides us with the notations to produce models. Different diagrams are drawn to visualize a system from different perspectives, so a diagram is a projection into a system. The language is very rich, and carries with it many aspects of Software Engineering best practice.

Unit Assessment

Instructions

Answer the following questions.

vi. Define the following terms as applied to Unified Modeling Language:

(a) UML as a language for communication (Answer: section 2.1.4)

(b) Modeling (Answer: section 2.1.5)

vii. Describe four aspects of UML as a language (Answer: section 2.1.4)

viii. List four advantages of modeling (Answer: section 2.1.5)

ix. Describe the four principles of modeling (Answer: section 2.1.6)

x. List UML diagrams and show those which are static diagrams from the list (Answer: section 2.2.1)

xi. Differentiate between the following terms:

(a) sequence diagram and collaboration diagram (Answer: section 2.2.5)

(b) Use case diagram and use case description (Answer: section 2.2.4)

(c) Extent and include use case relationships (Answer: section 2.2.4)

xii. Clearly explain why is it necessary to have a variety of diagrams in a model of a system? (Answer: section 2.2.1)

xiii. With the aid of a diagram explain four notations used with a use case diagram (Answer: section 2.2.4)

Answers

i. Definition of the following terms as applied to Unified Modeling Language:

(a) UML as a language for communication (2 marks)

Since communicating technical issues with customers has proven to be equally problematic, graphical modelling can make people (technical and non-technical) understand the artifacts of the software system expected. In software development, some of the things that require communication include requirements, design, implementation, and deployment. UML is a language designed to communicate these things

(b) Modeling (2 marks)

Modelling is the designing of software applications before coding (implementation in a particular programming language). A model is a representation or simplification of reality. It provides a blueprint of a system.

ii. Describe four aspects of UML as a language (@ 2 marks)

(a) The UML as a Language for Visualizing:

For many programmers, the distance between thinking of an implementation and then pounding it out in code is close to zero. You think it, you code it. In fact, some things are best cast directly in code. Text is a wonderfully minimal and direct way to write expressions and algorithms

(b) The UML as a Language for Specifying:

Specifying means of building models that are precise, unambiguous, and complete. In particular, the UML addresses the specification of all the important analysis, design, and implementation decisions that must be made in developing and deploying a software-intensive system.

(c) The UML as a Language for Constructing:

The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages. This means that it is possible to map from a model in the UML to a programming language such as Java, C++, Visual Basic or PHP, or even to tables in a relational database or the persistent store of an object-oriented database.

(d) The UML as a Language for Documenting:

A healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include (but are not limited to): Requirements, Architecture, Design, Source code, Project plans, Tests, Prototypes, and Releases. The UML addresses the documentation of a system's architecture and all of its details. The UML also provides a language for expressing requirements and for tests.

Finally, the UML provides a language for modelling the activities of project planning and release management.

iii. Four advantages of modeling (2 marks, @ 0.5 marks)

- To facilitate communication the desired structure and behavior of our system.
- To create visualization and control of the system's architecture.
- Enables better understanding of the system we are building, often exposing opportunities for simplification and reuse.
- Modelling Manages risk.
- Modelling manages Complexity
- Modelling Promotes Reuse
- Ensures business functionality is complete and correct,
- Ensures end-user needs are met, and
- Ensures program design meets requirements for scalability, robustness, security, extensibility, and other characteristics.

iv. List four principles of modeling (@ 1 mark)

(a) The choice of what models to create has a profound influence on how a problem is

attacked and how a solution is shaped.

(b) Every model may be expressed at different levels of precision.

(c) The best models are connected to reality.

(d) No single model is sufficient. Every nontrivial system is best approached through a small

set of nearly independent models.

v. List UML diagrams and show those which are static diagrams from the list (6 marks)

UML Diagrams:

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram
- Collaboration diagram
- Statechart diagram
- Activity diagram
- Component diagram

i. Deployment diagram

Static diagrams are:

- Class Diagram
- Object Diagram
- Component Diagram
- Deployment Diagram

vi. Differentiate between the following terms:

(d) sequence diagram and collaboration diagram (3 marks)

Sequence diagram

A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages. Sequence diagrams focus on the order in which the messages are sent. They provide a sequential map of message passing between objects over time

Collaboration diagram

A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Collaboration diagrams express both the context of a group of objects and the interaction between these objects. It focuses upon the relationships between the objects.

(e) Use case diagram and use case description (3 marks)

Use case diagram

Use case diagrams address the static use case view of a system. The different types of people and/or devices (called actors) that interact with the system are identified along with the functions that they perform or initiate. A Use Case diagram shows a set of use cases and actors (a special kind of class) and their relationships. These diagrams are especially important in organizing and modelling the behaviors of a system

Use case description

A scenario is a description of a use case as a story. It is a story of one user's interaction with the system. It can be described in terms of:

- The actors involved, and
- The steps to be taken in order to achieve the functionality described in the use-case title. (Use cases are about functionality, so make sure the title reflects some functionality.)

(f) Extends and includes use case relationships (3 marks)

"Extends" use case relationship

The "extends" notation extends the functionality of a use case to deal with errors or exceptions. "extends" relationship is being used when there is one use case that is similar to another but does a bit more.

"includes" use case relationships

"include" relationship can be used for making up a big use case from simpler ones.

vii. Clearly explain why is it necessary to have a variety of diagrams in a model of a system? (2 marks)

UML has a lot of different diagrams (models), the reason for this is that it is possible to look at a system from different viewpoints.

viii. With the aid of a diagram explain four notations used with a use case diagram (8 marks)

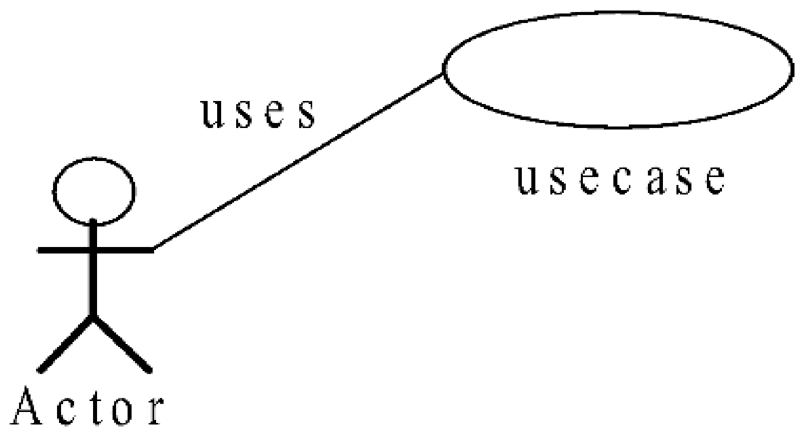
Four notations used with use case diagram are:

(a) Basic Use Case Notation

The Actor represents a user of the system, or any external system that interacts with the system. The Usecase represents a piece of functionality that is important to the user. Mostly we see the

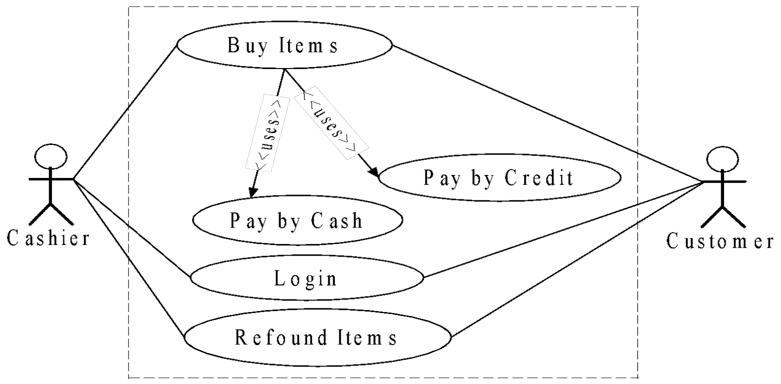
actor as a human user, but it can also represent a system or other nonhuman artifact.

Figure 2.4 shows the basic notation of a use case diagram.



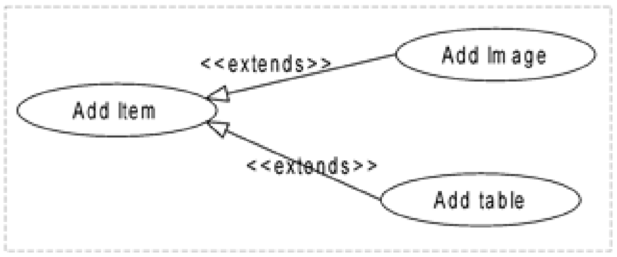
(b) Using the <<uses>> Relationship

This feature encourages re-use. If a use-case needs the functionality of another use-case in order to perform its task, it "uses" the 2nd use case. The relationship is drawn as a line with arrowhead pointing to the use case that is being "used" as shown in Figure 2.5



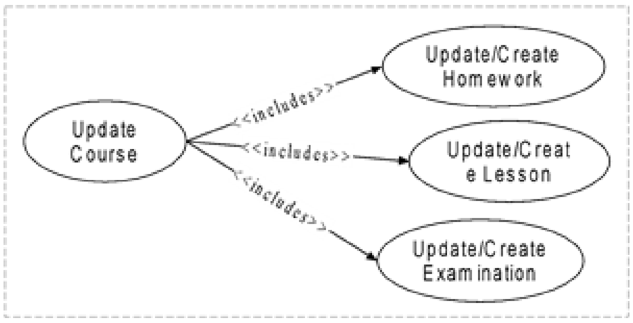
(c) Using the <<extends>> relationship

The "extends" notation extends the functionality of a use case to deal with errors or exceptions. "extends" relationship as shown in Figure 2.6 is being used when there is one use case that is similar to another but does a bit more. The relationship is drawn as a line with arrowhead pointing to the major use case.



(d) Using the << include >> relationship

"include" relationship as shown in Figure 2.7 can be used for making up a big use case from simpler ones.



Readings and Other Resources

- Ariadne Training (2001), "UML Applied Object Oriented Analysis and Design Using the UML", Ariadne Training Limited
- Booch G., Rumbaugh J., and Jacobson I., (2005), "The Unified Modeling Language User Guide 2nd Edition, ISBN: 0-201-57168-4, 512
- Larman C. (2004), "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", (3rd Edition) 3rd Edition, Prentice Hall; 3 edition (October 30, 2004), ISBN-13: 978-0131489066
- Liu Z., (2001), "Object-Oriented Software Development Using UML", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.
- Ojo A. and Estevez E., (2005), "Object-Oriented Analysis and Design with UML", Training Course, The United Nations University, UNU-IIST International Institute for Software Technology, e-Macao Report 19, Version 1.0, October.

Unit III: Object-Oriented analysis

Unit Introduction

This unit explains how requirement specifications can be obtained by involving use cases, identified classes and considering system operations and operation contracts which are obtained from system sequence model.

Unit Objectives

Upon completion of this unit you should be able to:

1. Produce functional specification, and non-functional attributes of systems
2. Model use cases, use case diagrams, and use-case model for describing the functional requirements
3. Explain use-case diagrams and their use for describing a use-case model

KEY TERMS

Analysis

Analysis emphasizes an investigation and understanding the problem domain and requirements, rather than a solution. For example, if a new computerized library information system is desired, how will it be used? "Analysis" is a broad term, best qualified, as in requirements analysis (an investigation of the requirements) or with reference to object-oriented approach as object analysis (an investigation of the domain objects).

Design

Design emphasizes a conceptual solution that fulfills the requirements, rather than its implementation. For example, a description of a database schema and software objects. Ultimately, designs can be implemented. As with analysis, design term is best qualified as in object design or database design. Analysis and design have been summarized in the phrase do the right thing (analysis), and do the thing right (design).

Use Case

A use case specifies a sequence of actions that the system can perform and that yield an observable result of value to a particular actor.

Activity 1 – What are Requirements

Introduction

Requirements as defined by Liu (2001) are a description or statement of a function, feature or condition that a user seeks to have implemented in a system. A requirement is:

1. A function that a system must perform
2. A feature of the system or a description of something the system is capable of doing in order to fulfil the system's purpose.
3. A statement about the proposed system that all stakeholders agree that must be true in order for the customer's problem to be adequately solved.

The creation of correct and thorough requirements specification is essential to a successful project. We use a case study (Point-of-Sale) to illustrate the following three problems.

1. What should be produced in the requirements capture and analysis?
2. How to identify the elements of these artifacts?
3. How are artefacts expressed?

Requirement Specification

Requirements documentation is a very important activity, which is written after the requirements elicitation/gathering and analysis, i.e. requirement capture and analysis phase of the software development process. This is the way to represent requirements in a consistent format. The requirements document is called the Software Requirements Specification (SRS). The SRS is a specification for a particular software product, program, or set of programs that perform certain functions in a specific environment.

The requirements specification is a description of needs or desires for a product. It is the official statement of what is required of the system developer. The requirements must be described unambiguously and in a form that clearly communicates to the client and to the development team members.

It is not a design document and it must state what to be done rather than how it is done. It must be in a form which can be taken as the starting point for the software development. A specification language is often used. Graphical notations are often also used to describe the requirement specification.

It is recommended that the requirements specification at least include the following information (Liu, 2001):

i. An overview of the project

ii. Goals/purpose

iii. Glossary – definition all relevant terms

iv. System functions

- System functional requirements, these are functional requirements
- System attributes (non-functional requirements)

v. Use cases – narrative descriptions of the domain processes

vi. Conceptual model – a model of important concepts and their relationships in the application domain.

These are typically produced through gathering and digesting

i. Documented information such as the client's statements about their needs, preliminary investigation report, and electronic documents,

ii. interview results,

iii. requirements definition meetings, and so on

An Overview of the Project

This is a sentence which gives the purpose of the project. This should describe the need for the system. e.g.

The project intends to create a standalone system for ………………

Goals

The purpose of this project is to create a point-of-sale terminal system to be used in retail sales. This is to describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.

The goals of the POST system can be stated as (Liu, 2001):

In general the goal is increased checkout automation, to support faster, better and cheaper services and business processes. More specifically, these include:

- Quick checkout for the customer,
- Fast and accurate sales analysis,
- Automatic inventory control.

Note: The overview and the goals can be combined into an introduction section in the document.

Glossary

Glossary is the definition of all relevant terms and concepts. It provides the clarification of ambiguous terms and concepts, explanations of jargons, as well as the description of business events or description of software actions. There is no specific format for glossaries and they can be included when needed.

System Functions

Requirements are the capabilities to which the system has to conform. Requirements are captured in terms of the system functions; hence system functions are requirements the system is supposed to do. There are two broad categories of requirements:

- Functional Requirements

  Are those that relate directly to the functioning of the system.

  These are the aspects of the system the client is most likely to recognize.

- Non-functional requirements:

  They are constraints/restrictions imposed on the system

Functional requirements

Functional requirements describe interaction between a system and its environment. Describe how a system should behave under certain stimuli. Liu, (2001) argue that to verify that some X is indeed a system function, it should make sense in the following sentence:

"The system should do < XX>"

The system's function should be categorized in order to prioritize them or to avoid from missing them. Categories include (Liu, 2001):

1. Evident functions whereby the requirement will be performed and recognized by the user.

2. Hidden functions whereby the requirement will be performed by the system, but not visible to users. Examples of hidden functions can be "save information in a persistent storage mechanism". It has been noticed that hidden functions are often forgotten hence missed during the requirements gathering process.

System functions can be presented as shown in template given in Table 3.1. They should be divided into logical cohesive (interrelated/organized) groups. Each function should be given a reference number that can be used in other documents of the development. A function should be category in either "evident" or "hidden".

Table 3.1: Template for Presenting Functional Requirements

Ref #	Function	Category

Tables 3.2 and 3.3 show functional requirements for the POST system (Liu, 2001).

Table 3.2: Basic functions of the POST system

Ref #	Function	Category
R1.1	Record the underway (current) sale - the items purchased	evident
R1.2	Calculate current sale total	evident
R1.3	Capture purchase item information from a bar code using a bar code scanner or manual entry of a product code such as a universal product code (UPC)	evident
R1.4	Reduce inventory quantities when a sale is committed	hidden
R1.5	Log completed sales	hidden
R1.6	Cashier must log in with an ID and password in order to use the system	evident
R1.7	Provide a persistent storage mechanism	hidden
R1.8	Provide inter-process and inter-system communication mechanisms	hidden
R1.9	Display description and price of item recorded	evident

Table 3.3: Payment functions of the POST System

Ref #	Function	Category
R2.1	Handle cash payments, capturing amount tendered and calculating balance due.	evident
R2.2	Handle credit payments, capturing credit information from a card reader or by manual entry, and authorizing payment with the store's (external) credit authorization service via a modem connection.	evident
R2.3	Handle cheque payments, capturing driver's license by manual entry, and authorizing payment with the store's (external) cheque authorization service via a modem connection.	evident
R2.4	Log credit payments to the accounts receivable system, since the credit authorization services owes the store the payment amount.	hidden

Requirements are expected to have the following characteristics:

(a) Verifiable:

A requirements document often becomes part of the contract between a client and a system developer. It describes what the developer will deliver and be paid for. Requirements should be verified against requirements, i.e. to check if the software meets the requirements.

(b) Complete

The requirements should be complete, and not leave out any areas. This can be very hard to achieve, especially in large systems. Current trends are to break the development up into smaller modules and implement each of them.

(c) Unambiguous

A requirement is ambiguous if it has more than one possible meaning. May be ambiguous due to poor choice of words and/or differing definitions of a word or phrase.

(d) Consistent

All parts of the requirements document should be consistent with each other. They should not contradict each other (the following do: “If you delete a task all of its subtasks should be deleted automatically” “You should always be prompted to confirm the deletion of a task and any of it’s subtasks”).

(e) Modifiable

The requirements should be structured so that it is possible to change it at little to no cost. Requires it be structured carefully, e.g. separating functional from non functional requirements, supplying a glossary (a table of definitions)

(f) Traceable

The requirements should be structured so that it is possible to uniquely identify each requirement

- Each one should have a unique number
- This allows it to be referred to in final testing of the delivered software, and in discussions with a client.

(b) Non-Functional Requirements

System non-functional requirements of system, also known as system attributes, are constraints imposed on the services or functions offered by the system. System attributes define how a system is supposed to behave and they are often called qualities of the system. They are being applied to the system as a whole. Examples are; usability, cost, reliability, response time, performance, ease of use, security, compatibility, fault-tolerance, and so on

You can’t write a specific line of code to implement non-functional requirements; rather they are “emergent” properties that arise from the entire solution.

Table 3.4 shows examples of non-functional requirements for the POST system.

Attribute	Constraints
response time	When recording a sold item, the description and price will appear within 5 seconds
interface metaphor	Forms-metaphor windows and dialog boxes
fault-tolerance	Must log authorized credit payments to accounts receivable within 24 hours, even if power or device fails
operating system platform	Microsoft Window 95 and NT

Non-functional can be classified into (Sommerville, 2000)::

(a) Product requirements

Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

(b) Organisational requirements

Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.

(c) External requirements

Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Conclusion

Functional requirements are requirements in which users seek for them to be implemented. However, there are also requirements done by the system, but not directly recognized by users, these requirements should be considered when documenting requirements. These are the once referred to hidden functionalities.

Activity Assessment

With respect to students' case projects, clearly do the followings:

- Provide an overview and goals of your system you wish to develop

- List the core functional requirements of the system you wish to develop. For each requirement listed, provide the requirement identifier, description, and cross references
- List five nonfunctional requirements of the system using the format specified

Activity 2 – Use Case

Introduction

This activity explains one of the powerful components in capturing system functions. This is a use case, whereby, apart from having use cases, we still need to represent them in view of being interacted from the external environment; this is a use case diagram. A use case being a system function, need to be described for the purpose of understanding how the function is going to be performed. In this activity you are going to identify use cases, draw a use case diagram and describe use cases.

Use case concepts

A use case is a very powerful UML tool. A use case describes and captures functional requirements, so uses cases are requirements or are functional requirements that indicate what the system will do. They also describe a set of interactions between a user and the system. A set of use case can be a description of the entire system to be implemented.

For example, to carry out the process of buying things at a store when a POST case study is used (Liu, 2001)

- two actors must be involved: Customer and Cashier,
- the following sequence of events must be performed:

√ The Customer arrives at a checkout with items to purchase.

√ The Cashier records the purchase items and collects payment.

√ On completion, the Customer leaves with the items

To a user, a use case is a way of using the system. When a user is interacting with a system, system provides a service to the actors. Each use case then captures a piece of functional requirements for some users. All the use cases together describe the overall functional requirements of the system. The first step in requirement capture is to capture requirements as use cases. UML provides a simple notation to represent a use case as follows:

Actors

A Use case cannot initiate actions on its own. An actor is someone who can initiate a Use Case. An actor is representing a certain role, instead of representing a particular individual. An actor can be users or external systems that the system interacts with.

With Liu, (2001) an actor:

- Stimulates the system with input events, or receives something from the system.
- Communicate with the system by sending messages to and receiving messages from the system as it performs use cases.
- May model anything that needs to interact with the system to exchange information. Actors can be human users, computer systems, electrical or mechanical devices such as timers.

If there is more than one actor in a use case, the one who generates the starting stimulus is called the initiator actor and the other participating actors. The actors that directly interacts the system are primary/direct actors, the others are called secondary actors.

Example:

An auto bank machine system (ATM system) (Case Study 2 under unit 0) interacts with a type of users who will use the system to withdraw money from accounts, to deposit money to accounts, and to transfer money between accounts. This set of users is represented by the Bank Customer actor. Therefore a use-case model can be used to represent the three use cases WithDraw Money, Deposit Money, and Transfer Money that has association to the Bank Customer actor as shown in Figure 3.2 (Liu, 2001).

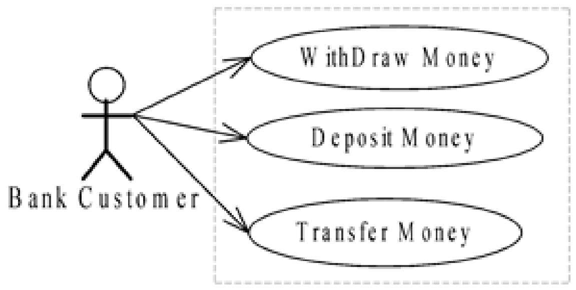


Figure 3.2: The Relationship of an Actor and a Use Case

Thus, actors represent parties outside the system that collaborate with the system. Once we have identified all the actors of a system, we have identified the external environment of the system.

For most systems, a single actor can interact with many use cases, and a single use case can be initiated by many different actors. Taking an example from the POST case study, Figure 3.3 shows a complete system described using actors and use cases.

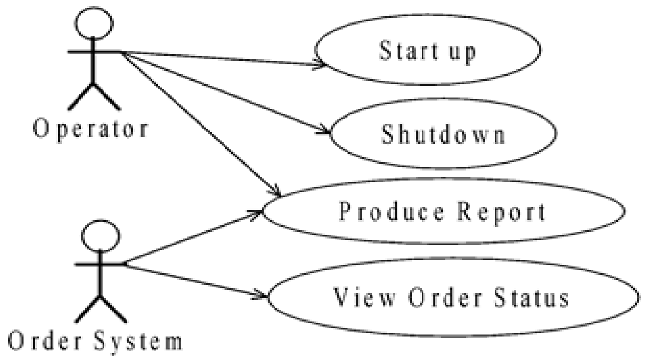


Figure 3.3: Use Case Interacted by two Actors

Use Case Granularity

Ariadne (2001) elaborate clearly that it can be difficult to decide upon the granularity of use cases in particular scenario, should each user-system interaction be a use case, or should the use case encapsulate all of the interactions? Taking an example of the ATM system in Figure 3.3 to allow a user to withdraw money, we may have the following interactions:

- Enter card
- Enter pin number
- Select amount required
- Confirm amount required
- Remove card
- Take receipt

Should each of these steps e.g. "enter pin number" be a use case?

This is a classic mistake in construction of use cases. Here we have generated a large number of small, almost inconsequential use cases. In any non-trivial system, we would end up with a huge number of use cases, and the complexity would become overwhelming (Ariadne, 2001).

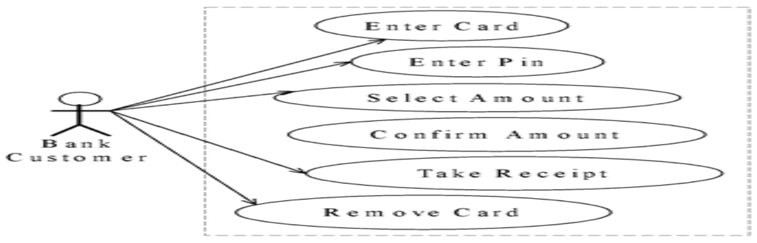


Figure 3.4: A Not Useful Use Case Diagram

To handle the complexity of even very large system, we need to keep the use cases at a fairly "high level". The best way is to keep the following rule-of-thumb in mind. (Ariadne, 2001).

"A Use Case should satisfy a goal for the actor"

If we apply this simple rule to our example above, we can notice than none of the presented use cases in Figure 3.4 is qualifying to be a use case, since they do not describe the goal of the use case. The goal of use cases could be withdraw money, deposit money etc as shown in figure 3.5.

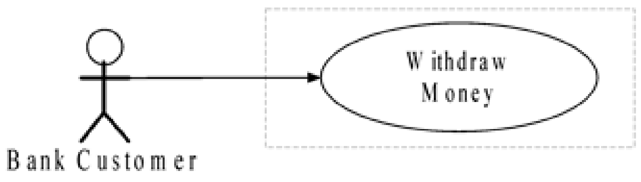


Figure 3.5: A More Focused Use Case Diagram

Use Cases Identification

Use cases can be identified in two methods: actor-based and event based. Functional requirements document is the main source of information needed. Interviews with potential users of the system can also be one source of getting use cases. In some cases Joint Requirement Planning Workshops (JRP) can be used where all people interested in system being developed come together to give their view of what the system needs to do. Liu, (2001 state that identifying use cases involves reviewing existing documents of requirements specification:

i. Use case identification on actor-based: method

(a) Find and specify all the actors by looking at which users will use the system and which other systems must interact with it.

(b) For each actor, identifying the processes they initiate or participate in by looking at how the actor communicate/interact with (or use) the system to do his work.

ii. Use case identification on event-based method

(a) Identify the external events that a system must respond to.

(b) Relate the events to actors and use cases.

For the POST application the following potential use actors and use cases as in Table 3.5 can be identified (Liu, 2001) and the two can be presented in a use case diagram as shown in Figure 3.6:

Table 3.5: Actors and Use Cases of POST System

Actor	Processes to Initiate
Cashier	Log In, Log Out, Cash Out
Customer	Buy Items, Refund Items
Manager	Start Up, Shut Down
System Administrator	Add New User

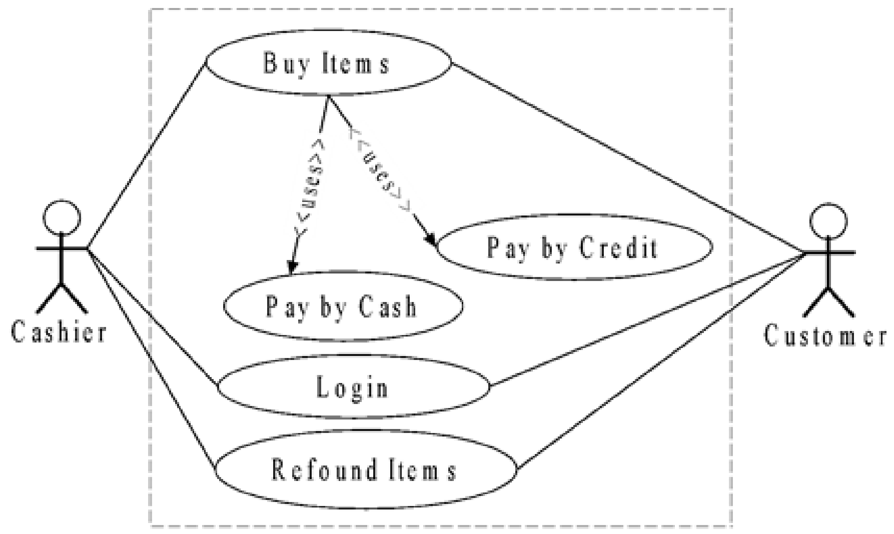


Figure 3.6: Partial Use Case Diagram for POST System**Use Case descriptions**

Each Use Case contains a full set of textual details about the interactions and scenarios contained within it. Use cases are described in order to understand more on its functionality. Table 3.6 shows the template for a Use Case description:

Table 3.6: A Template for a Use Case Description

Use Case:	Use Case name
Actors:	Role names of people or external entities initiating and participating in the use case
Short Description:	A brief description of the Use Case
Pre-Conditions:	A description of the conditions that must be satisfied before the use case is invoked, i.e. what must always be true before beginning a use case scenario.
Post-Conditions:	A description of what has happened at the end of the use case, i.e. what must be true on successful completion of a use case.
Main Flow:	A list of the system interactions that take place under the most common scenario. For example for "withdraw money", this would be "enter card, enter pin, etc"

Alternate Flow)s):	A description of possible alternative interactions
Exception Flow(s):	A description of possible scenarios where unexpected or unpredicted events have taken place

In some cases, a high-level use case can be needed to obtain some understanding of the overall process, and then expand it by adding to it with more details. A high-level use case describes a process very briefly, usually in two or three sentences.

Table 3.7: A Template for High-Level Use Case

Use case:	Name of use case (use a phrase starting with a verb).
Actors:	List of actors (external agents), indicating who initiates the use case.
Purpose:	Intention of the use case.
Overview:	A brief description of the process.
Cross References:	Related use cases and system functions.

With reference to a POST case study, the following as shown in Table 3.8 can be a description of a "Pay by Cash" use case

Table 3.8: A "Pay by Cash" Use Case Description

Use Case:	Pay by Cash
Actors:	Customer, Cashier
Short Description:	A Use Case allows a cashier to record the cash tendered by the customer
Pre-Conditions:	Items to be purchased listed and the total amount already displayed

Post-Conditions:	A customer will be presented with a printed receipt and the items purchased
Main Flow:	1. The Customer presents cash payment to the cashier.
	2. The Cashier records the cash tendered to the POST system.
	3. The system shows the balance due back to the Customer if any.
	4. The Cashier deposits the cash received and extracts the balance due.
	5. The Cashier gives the balance due and the printed receipt to the Customer.
Alternate Flow)s):	Item 4: Insufficient cash in drawer to pay balance. Ask for cash from supervisor.
Exception Flow(s):	Item 3. If cash amount tendered is not enough, exception handling

Having been identified and described use cases, a use case model can be created which describes how use cases relate to each other and to the actor. A use case diagram describes part of the use-case model and illustrates a set of use cases for a system, the actors, and the relation between the actors and use cases

Use cases relationships such as include, uses and extend can be used to reduce the complexity of the model.

Use cases within a development process

Activities in the use case analysis can be summarized as follows:

1. After system functions have been listed, then identify actors and use cases.
2. Draw a use case diagram for the system.
3. Relate use cases and illustrate relationships in the use case diagram.
4. Write the most critical, influential and risky use cases in the use case high-level
5. Describe the uses cases using a use case description template

Identified Actors and Their Related Use Cases for the POST system

Applying the techniques used to identify use cases, we can have a sample list (not an exhaustive) of relevant actors and use cases as shown in Table 3.9:

Table 3.9: Actors and Use Cases for POST System

Actor	Use Case
Cashier	Log In / Logout
	Cash Out
Customer	Buy Items: uses
	• By cash
	• By credit
	Refund Items
Manager	Start Up
	Shut Down
System Administrator	Add New Users

Why use cases?

The reasons why use cases are good for requirement capture include:

1. They answer what the system should do for each user or actor question.

2. Use Cases define the scope of the system. They enable to visualize size and scope of the entire development.

3. They facilitate communication means between the customer and developers (use case diagram is so simple, anyone can understand it)

4. They represent system functions and the relationship among the system functions in a narrative way.

5. They extent the identification of concepts and objects involved in the application domain.

6. Use Cases are very similar to requirements, but whilst requirement tend to be vague, confusing, ambiguous and poorly written, the tighter structure of Use Case tend to make them far more focused

7. The "sum" of the use cases is the whole system. Anything not covered by a use case is outside the boundary of the system to be developed. So the Use Case diagram is complete with no holes
8. Use Cases guide the development teams through the development process. They give the summary of what is needed to be done by the system
9. Use Cases provide a method for planning development work, and allow to estimate how long the development will take
10. Use Cases provide the basis for creating system tests
11. Use cases help with the creation of user guides
12. Use cases can be a source of information during validation and verification of a software

Conclusion

Use cases are a powerful way of modelling what the system needs to do. A use case is full end-to-end story about the use of the system to carry out a task, not an arbitrary combination of a number of steps of computation. Use cases are excellent way of describing the system's scope. We need to be careful on the granularity of the Use Cases to control complexity.

Activity Assessment

i. With reference to use case relationship, when would you use the followings and why?

(a) <<extends>>

(b) <<uses>>

ii. Explain why the task of the requirements capture and analysis is difficult and complex

iii. How do system functions and use cases relate?

iv. Differentiate between system functions and use cases

v. Use the following given library system to answer questions.

Assume that a member of the library is allowed to borrow up to six items (e.g. a copy of a book, a journals, etc). The system supports to carry out the tasks of borrowing a copy of a book, extending a loan, and checking for reservation.

(a) Identify the use cases that represent these tasks,

(b) Draw a use-case diagram to represent relationships among these uses cases.

vi. Using individual/group case projects, construct the use cases by doing the following:

(a) Identify the use cases for your system

(b) Identify the actors related to each of the use cases identified in (a)

(c) Select only four key use cases (in task (a) which reflect the main tasks of your project for demonstration and write use case descriptions for these use cases.

(d) Draw a use case model to show the relationships between actors and all use cases for your system

Note: Keep in mind that the four key selected use cases will be used in the following stage of software development process.

Activity 3 – Conceptual Modelling

Introduction

A central distinction between OOA and structured analysis is decomposition by concepts (objects) rather than decomposition by functions. The key point in this decomposition is to find what can behave, and then to decide later on in the design and implementation how they behave to realize the system functionality. Object-oriented requirement analysis is more concerned in identifying concepts related to the requirements and to create a conceptual model of the problem domain or the application area we are working with. This activity is will guide you on how to identifying objects or concepts and make them relate to each other and finally create a conceptual or domain diagram.

Concepts and Classes in Conceptual Modelling

Conceptual Modelling (sometimes called Domain Modelling) is the activity of finding out which concepts are important to system. This process helps us to understand the problem further, and develop a better awareness of our customer's business. A concept is an idea, thing, or object. More formally, a concept may be considered in terms of its symbol, intension, and extension (Liu, 2001):

- Symbol – words or images representing a concept. It can be referred to when we talk about the concept.
- Intension – the definition of a concept.
- Extension – the set of examples or instances to which the concept applies.

For an example, the symbol Module as applied in some Universities can have:

- The intension to "represent a course offered as part of a degree in that university"; and
- The extension of all other modules offered by that university.

Note that the terms class and type are used more by UML and not concept. As long as we are in requirement analysis stage, the two terms class and concept may be used interchangeably.. Each instance of a class is called an object of the class. For example a class called "Student". JahnSmith and JaneBrown are instances of a class Student. Therefore, a class defines a set of objects.

The notions of class and object are interlinked as one cannot exist without the other, and any object belongs to a class. The differences are:

- An object is a concrete entity – exists in space and time (persistence property of objects);
- A class is an abstraction of a set of objects.

The UML defines the term class as "a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics".

Identifying Concepts

The requirement document which is up to now including: the overview, goal, functional requirements, use cases, use case diagram and use case descriptions is the good source of information in identifying concepts. Two activities that are fundamental to the useful application of object oriented in OOA are:

- To identify as many candidate objects from the problem domain, and
- Later select those candidate objects that are significant to develop and be specified in an object model (conceptual model)

Liu (2001) suggests the following candidate concepts from the requirement document as shown in Table 3.10.

Table 3.10: Guide to Identify Concepts

Concept Category	Examples
physical or tangible objects (or things)	POST (i.e. Point-of-Sale Unit), House, Car, Sheep, People, Airplane
places	Store, Office, Airport, PoliceSation

documents, specifications, designs, or descriptions of things	ProductSpecification, ModuleDescription, FlightDescription
transactions	Sale, Payment, Reservation
roles of people	Cashier, Student, Doctor, Pilot
containers of other things	Store, Bin, Library, Airplane
things in a container	Item, Book, Passenger
other computers or electro-mechanical systems external to our system	CreditCardAuthorizationSystem, Air-TrafficControl
abstract noun concepts	Hunger, Acrophobia
organizations	SalesDepartment, Club, ObjectAirline
historic events, incidents	Sale, Robbery, Meeting, Flight, Crash, Landing
Processes (often not represented as a concept, but may be)	SellingAProduct, BookingASeat
rules and policies	RefundPolicy, CancellationPolicy
catalogs	ProductCatalog, PartsCatalog
records of finance, work, contracts, legal matters	Receipt, Ledger, EmploymentContract, MaintenanceLog
financial instruments and services	LineOfCredit, Stock
manuals, books	EmployeeManual, RepairManual

Another useful and simple technique for the identification of concepts is to identify or extract noun and noun phrases from the requirement document and consider them as candidate concepts or attributes.

Warning:

Care must be applied when these methods are used; mechanical noun-to-concept mapping is not possible, words in natural languages are ambiguous, and concept categories may include concepts which are about either attributes, events or operations which should not be modeled as classes. We should concentrate on the objects/classes involved in the realization of the use cases (Liu, 2001).

Consider the POST system, from the use case Buy Items with Cash. We can identify some noun phrases as shown in Figure 3.8. Some of the noun phrases in the use case are candidate concepts; some may be attributes of concepts (Liu, 2001)

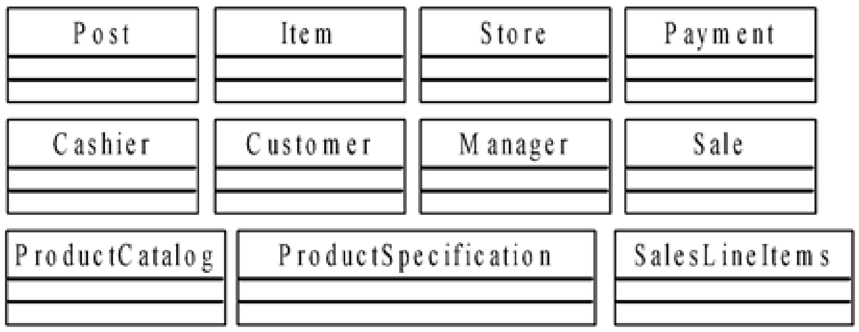


Figure 3.8: Selected Possible Concepts for the POST System

The following guidelines are useful when identifying concepts (Liu, 2001):

- It is better to over specify a conceptual model with lots of fine-grained concepts, than to under specify it.
- Do not exclude a concept simply because the requirements do not indicate an obvious need to remember information about it.

- It is common to miss concepts during the initial identification phase, and to discover them later during the consideration of attributes or associations, or during the design phase. When found, they are added to the conceptual model.
- Put a concept down as a candidate in the conceptual model when you are not sure it must not be included.

Adding Associations in a Conceptual Model

A conceptual model with totally independent concepts only is obviously useless, as objects in different classes must be related to each other so that they can interact and collaborate with each other to carry out processes. In UML, an association is a relationship between two classes that specifies how instances of the classes can be linked together to work together.

In the same sense that instances of a class are objects, instances of an association are links between objects of the two classes – this is what we meant that objects in the same class "share the same relationships".

With respect to an association between classes, an important information is about how many objects of one class (say "A") can be associated with one object of another (say "B")n , at a particular moment in time. We use multiplicity to represent this information. An example in Figure 3.9 shows the multiplicity expressions and their meanings.

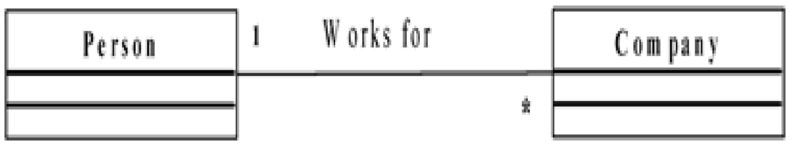


Figure 3.9: An Example on Multiplicity Use

Some high priority associations useful to include in a conceptual model are (Liu, 2001):

- A is a physical or logical part of B.
- A is physically or logically contained in/on B.
- A is recorded in B.

From the list of concepts identified, one can start thinking on the possible associations concepts have to each other. This can accelerate the creation of a conceptual diagram. In this way relevant associations will not be forgottern. Figure 3.10 shows possible aggregation relationships of concepts in a POST system.

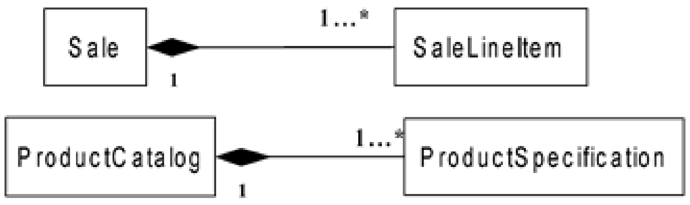


Figure 3.10: Aggregation Association in POST ApplicationAs an example, a conceptual model, or sometimes known as a class diagram for a POST system domain is as shown in Figure 3.11. This has also considered concepts' association: inheritance, aggregation and composition as discussed earlier.

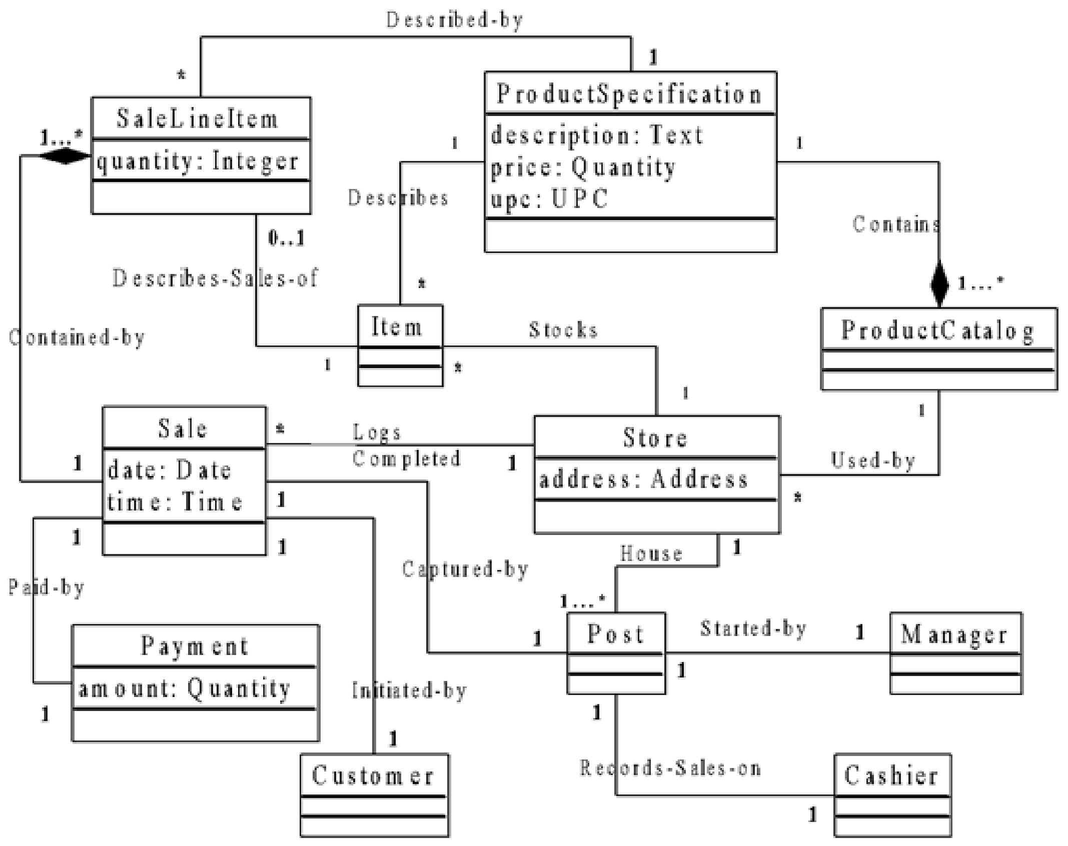


Figure. 3.11: A Conceptual/Class model for the POST System (concepts, associations and Attributes)

Add Attributes in a Conceptual Model

Instances of a concept may have some properties. For examples, a Sale can have a date and time; a Payment can have amount, a Furniture can have a size, a Module can have a code, title, and number of credits, a Student has a name, registration number and age, etc.

An attribute of a class is the abstraction of a single characteristic or a property of entities that have been abstracted as objects of the class. At any given moment of time, the attribute of an individual object in the class is a logical data value representing the corresponding property of the object, and called the value of attribute for the object at that time. One object has exactly one value for each attribute at any given time. Therefore, the value for an attribute of an object may change over time. For examples (Liu, 2001),

- time and date are attributes of class Sale, and an instance of Sale can be a sale at 13.30 on 1/10/1998.
- code, title, and credit are three attributes of class Module, and an instance of Module can have a code MC 206, title: Software Engineering and System Development, and credit: 20.
- name and age are attributes of Student, an individual student can have the name John Smith, and age 19.

Something to note in attributes:

(a) The attributes in a conceptual model should be simple and clear without ambiguity. Examples of simple attributes are Date, Number, PhoneNumber, Name, Description, Code, Title etc

(b) In object-oriented approach no foreign keys. Attributes are not be used to relate concepts in the conceptual model, but to store some information about the objects themselves. Liu, (2001) state that the most common violation of this principle is to add a kind of foreign key attribute, as is typically done in relational database designs, in order to associate two types. For example, the currentPOSTNumber attribute in the Cashier type in Figure 3.12 is undesirable because its purpose is to relate the Cashier to a POST object. The better way to express that a Cashier uses a POST is with an association, not with a foreign key attribute Once again

Relate types with an association, not with an attribute

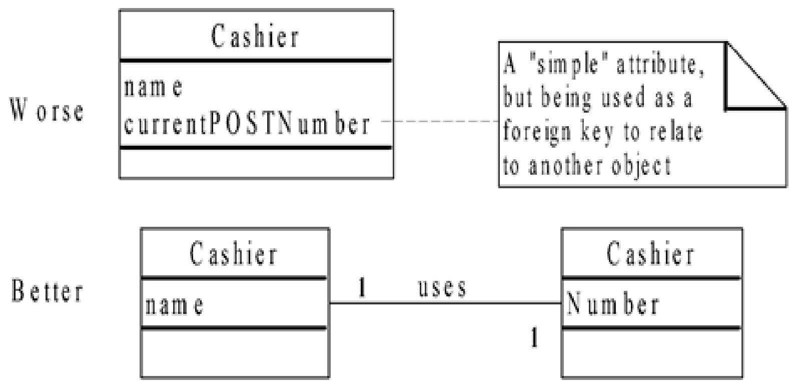

Worse
Cashier
name
currentPOSTNumber
A "simple" attribute, but being used as a foreign key to relate to another object
Better
Cashier
name
1
uses
1
Cashier
Number

Figure 3.12: Do not use attributes as foreign keys

Steps to Create a Conceptual Model

The following steps can be applied to create a conceptual model:

- Identify and List candidate concepts using the Concept Category List and noun phrase identification related to the current requirements under consideration.
- Represent them in a conceptual model.
- Add associations necessary to record relationships for which there are a need to preserve some memory.
- Add the attributes necessary to fulfill the information requirements.

Conclusion

System concepts can be identified by investigating the requirement document which includes: system function, use cases and other initial reports on the domain. A conceptual model show the static view of associations of concepts, they include as shown in Figure: 3.8 above; Concepts, Relationship or association between concepts and Attributes of concepts.

The following elements are not suitable in a conceptual model (Liu, 2001):

- A software artifact, such as a window or a database, unless the domain being modeled is of software concepts, such as a model of a graphical user interface.
- Operations (responsibilities) or methods.

Activity Assessment

i. Read the following problem and answer questions

A weather station is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected every five minutes. Weather stations transmit their data to the area, computer in response to a request from that machine. The area computer collates the collected data and integrates it with reports from other sources such as satellites and ships. It then generates a set of local weather maps.

(a) Identify classes or concepts from the given scenario

(b) Identify attributes of the concepts identified

(c) Draw a conceptual diagram of the problem domain.

ii. Consider the following problem domain:

"The bank client must be able to deposit the amount to and withdraw the amount from his/her account using touch screen. Each transaction must be recorded, the client must be able to review all the transactions performed in the account. Record transactions must include date, type, amount, and account balance after the transaction. A client can have two types of account – a checking and a savings account. Access to the ATM account is provided by a PIN code of 4 integer digits from 0 to 9.

(a) Identify classes or concepts from the given scenario

(b) Identify attributes of the concepts identified

(c) Draw a conceptual diagram of the problem domain

iii. From individual/group case studies, do the following:

(a) Identify possible concepts/classes for your system

(b) Group concepts/classes identified in (a) above and find their relationships/ associations

(c) Create a class/conceptual diagram to relate these concepts identified in (b)

(d) Identify attributes for concepts/classes used in a class diagram.

Activity 4 – System Behaviour: System Sequence Diagrams and Operations

Introduction

We need to identify the operations that the system needs to perform and in what order the system need to perform these operations to carry out a use case, and the effect of such an operation on the system, i.e. on the objects of the systems.

A use case defines a class of conversations between the actors and the system, and an individual conversation of this class is a realization of the use case. Obviously, there may be many realizations for a use case. A scenario of a use case is a particular instance or realized path through the use case, i.e. a particular realization of the use case.

This activity is intending to introduce system sequence diagrams which are used to find system events and system operations. Latter we will see how to create a contract of system operations.

System operations are the operations that system needs to perform to carry out a use case and the effects of these operations on the system. In this case, use cases – use case diagram and use case descriptions are inputs of this activity.

System Input Events and System Operations

When an actor interacts with the system for a certain use case, events are being generated to a system. This event is requesting the system to perform some operations in response. Liu, (2001) indicate that events generated by actors are very tightly related to operations that the system can perform. This implies that we identify system's operations by identifying events that actors generate.

By definition: A system input event is an external input generated by an actor to a system. A system input event initiates a responding operation while a system operation is an operation that the system executes in response to a system input event.

Some system operations also generate output events to the actors to prompt the next system event that an actor can perform; and A system event is either an input event or an output event (Liu, 2001)..

Therefore, a system input event triggers a system operation, and a system operation response to a system input event.

As an example given by Liu, (2001), we can formally define a scenario of a use case as a sequence of system events that occur during a realization of the use case. Consider the use case of Make Phone Calls for a telephone system, which involves two actors, Caller (initiator) and Callee. The following sequence of events is a scenario of this use case:

Input Events	Output Events
1. Caller lifts a receiver	2. Telephone starts dial tone
3. Caller dials a number	4. Telephone rings to Callee
	5. Telephone is ringing tone to Caller
6. Callee answers the ringing phone	7. Telephone ringing tone stops
	8. Telephone ringing tone stops
	9. Telephones connected
10. Callee hangs up by returning the telephone receiver	11. connection broken
12. Caller hangs up by returning the telephone receiver	

We can represent the set sequence of events is a scenario using event tracer diagram as shown in Figure 3.13:

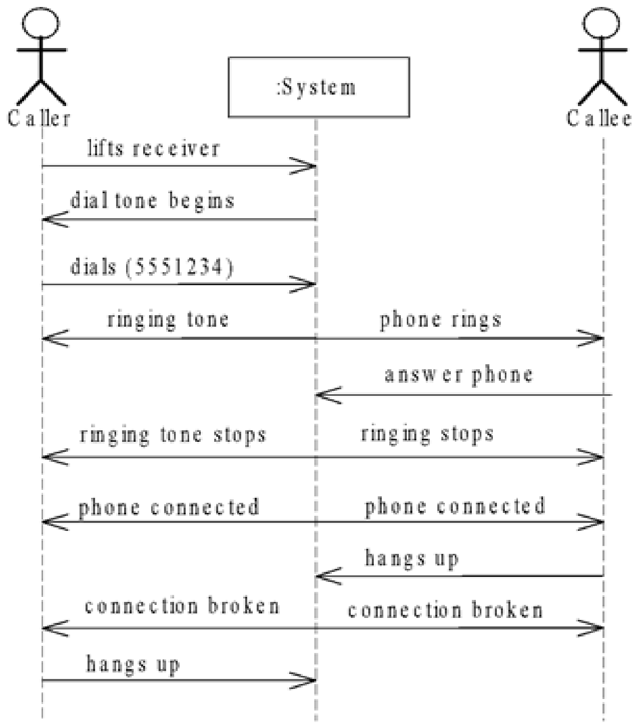


Figure 3.13: Telephone system event trace diagram for the "Make Phone Calls"

(Source: Liu, 2001)

A System Sequence Diagram

During requirements analysis phase, we define the system by treating it as a single black box so that the behaviour is a description of what a system does, without explaining how it does it. In this way, only system operations that an actor requests of the system will be considered. The interest is to find input events by making use of use cases and their scenario.

For example, the typical course of events in the Make Phone Calls indicates that the caller and callee generate the system input events that can be denoted as liftReceiver, dialPhoneNumber, answersPhone, hangsUp. In general, an event takes parameters (Liu, 2001).

UML Trace diagram is very useful in identifying the system operations, as in Figure 3.13 which show, for a particular course of events within a use case, the external actors that interact directly with the system, the system (as a black box), and the system input events that the actors generate. A simplified trace diagram which shows only system input events is called a system sequence diagram. A system sequence diagram (SSD) for the Make Phone Calls use case can be illustrated as in Figure 3.14.

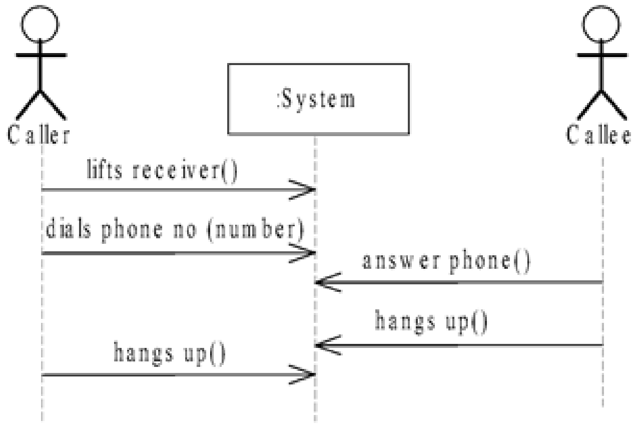


Figure 3.14: A System Sequence Diagram for the Make Phone Calls of a Telephone System

(Source: Liu, 2001)

Recording system operations

The set of all required system operations are determined by identifying the system input events. The name of the input event and the name of the operation are identical; the distinction is that the input event is the named stimulus, the operation is the response.

For example, from the inspection of the use case Buy Items with Cash, we can identify the system operations to be (Liu, 2001):

- enterItem(upc, quantity)
- endSale()
- makePayment()

The system operations can be grouped as operations of a type named System. The parameter may optionally be ignored. Figure 3.15 shows a system concept showing identified system operation.

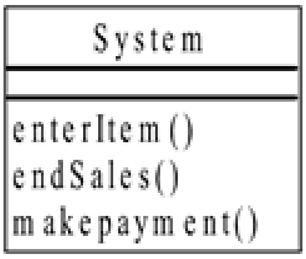


Figure 3.15: System Concept Showing System Operations for a “But Items with Cash” Use Case

The system operations for “Make Phone Calls” use case is shown in Figure 3.16.

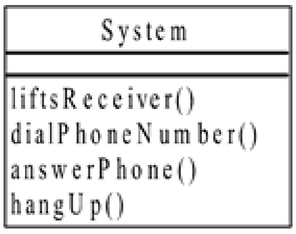


Figure 3.16: System Concept Showing System Operations for a “Make Phone Calls” Use Case

System Operations Contracts

Use cases describe system behavior which is usually sufficient. But, sometimes a more detailed description of system behavior is needed. Again system sequence diagram does not describe the effect of the execution of an operation invoked. It is missing the details necessary to understand the system response – the system behaviour. Part of the understanding of the system behaviour is to understand the system state changes carried out by system operations. A system state is a snapshot of the system at a particular moment of time which describes the objects of classes currently existing, current values of attributes of these objects, and current links between the objects at that time. The execution of a system operation changes the system state into another state (Liu, 2001):

- Old objects may be removed,
- New objects and links may be created, and
- Values for attributes of objects may be modified.

To describe detailed system behavior in terms of state changes to objects in the Domain Model, after a system operation has executed, contracts are being used. We need to know how to write a contract for a system operation. As explained by Liu, (2001) the contract of an operation is defined mainly in terms of its pre-conditions and post-conditions.

- Pre-conditions are the conditions that the state of the system is assumed to satisfy or must hold true before the execution of the operation
- Post-conditions are the conditions that the system state has to satisfy when the execution operation has finished, or the constraint that must hold true after the completion of an operation.

Documenting Contracts

Suggested schema by Liu, (2001) for presenting a contract is as shown in Table 3.11:

Table 3.11: Schema for Documenting a Contract of a System Operation

Name	Name of a system operation and parameters
Responsibilities	A short description of the responsibility the operation
Type	Name of type (concept, software class, interface)
Cross References	System function reference numbers, use cases, etc
Note	Design notes, algorithms, and so on
Exceptions	Exceptional cases
Output	Non-User Interface (UI) outputs, such as messages or records that are sent outside of the system
Pre-conditions	The conditions that the state of the system is assumed to satisfy before the execution of the operation
Post-conditions	Describes changes in the state of objects in the domain area. The sate changes can include: instances creation, associations formed or deletion and attributes changed.

Contracts for Some Operations in POST system

Tables 3.12 and 3.13 show examples contracts for "enterItem" operation and "makePayment" operation of the POST system.

Table 3.12: Contract for enterItem Operation Contract

Name	enterItem(upc:UPC, quantity:Integer):
Responsibilities:	Enter (or record) sale of an item and add it to the sale. Display the item description and price.
Type:	System.
Cross References:	System Functions: R1.1, R1.3, R1.9
	Use Cases, Buy Items:
Note:	Use superfast database access.
Exceptions:	If the UPC is not valid, indicate that it was an error.
Output:	
Pre-conditions:	UPC is known to the system.

Table 3.13: Contract for makePayment Operation

Name:	makePayment(amount: Quantity).
Responsibilities:	Record the payment, calculate balance and print receipt.
Type:	System.
Cross References:	System Functions: R2.1
	Use Cases: Buy Items
Note:	Use superfast database access
Exceptions:	• If sale is not complete, indicate that it was an error.
	• If the amount is less than the sale total, indicate an error.
Output:	
Pre-conditions:	

Post-conditions:	1. A Payment was created (instance creation).
	2. Payment.amountTendered was set to amount (attribute medication).
	3. The Payment was associated with the Sale (association formed).
	4. The Sale was associated with the Store, to add it to the historical log of completed sales (association formed).

How to Create a Contract

Apply the following advice from Liu, (2001) to create a contract for a system operation

1. Identify the system operations from the system sequence diagram.
2. For each system operation, construct a contract.
3. Start by writing the Responsibilities section, informally describing the purpose of the operation.
4. Then complete the Post-conditions section, declaratively describing the state changes that occur to objects in the conceptual model.
5. To describe the post-conditions, use the following categories:

- Instances creation and deletion.
- Attributes modification.
- Associations formed and broken.

Conclusion

Relating the three models used so far in this unit, it can be observed that:

i. Use cases suggest the system input events and system sequence diagram.

ii. The system operations are then identified from system sequence diagrams.

iii. The effect of the system operation is described in contract within the context of the conceptual model.

Activity Assessment

i. Discuss how post conditions of an operation can be generated from the use case

of the operation and the conceptual model. What are the relationships among

the three main kinds of effects of an operation described in the post-conditions?

ii. Clearly describe the importance of:

(a) System sequence diagram

(b) Operation contracts

iii. From individual/group case projects, do the following:

(a) Identify system input events for four selected key use cases which reflect the main

tasks of your project, with the help of system sequence diagram. These are associated

with system operations. Other operations for use cases can be presented in tabular

form. Note: consider using four Use Cases selected as a continuation.

(b) Create contracts for at least four system operations each obtained from each key use

case in (a)

Note: The system operations selected will be used in the object oriented design stage

UNIT SUMMARY

Object-oriented analysis phase is concerned with the understanding of requirements, concepts and system operations related to the system's area of application. A number of tools have been applied to do the analysis, including: use cases, use case diagrams, use case scenario, conceptual model, tracer sequence diagram, system sequence diagrams and system operation contracts. Investigation and analysis are often characterized as focusing on questions of what – what are the processes, concepts, associations, attributes, operations.

We have explored the following minimal but useful set of artifacts that can be used to capture the results of an investigation and analysis:

Analysis Artifact	Questions Answered
Use Cases	What are the domain processes?
Conceptual Model	What are the concepts, associations and attributes?
System Sequence Diagrams	What is the system input events and operations?
Contracts	What do the system operations do?

Unit Assessment

Instructions

Answer the following questions.

i. List characteristics of requirements and describe any three of them (Answer: section 3.1.2.4)

ii. Explain the following terms as applied to object-oriented analysis

(a) Requirement (Answer: section 3.1.1)

(b) Requirement specification (Answer: section 3.1.2)

iii. List six information to be included in a requirement specification (Answer: section 3.1.2)

iv. Differentiate the following terms:

(a) Functional requirements and non-functional requirements (Answer: section 3.1.2.4)

(b) Evident functions and hidden functions (Answer: section 3.1.2.4)

v. With an example explain the concept of use case granularity (Answer: section 3.2.2)

vi. Explain two ways applied to identify use cases (Answer: section 3.2.2)

vii. List six reasons why use cases are important (Answer: section 3.2.7)

viii. List three possible state change when a system operation is executed (Answer: section 3.4.5)

ix. Explain the importance of a contract on object-oriented analysis (Answer: 3.4.5)

x. How do “system input event” relate to “system operation” (Answer: section 3.4.2)

Answers

i.Requirements characteristics and their descriptions (Any three of them) (@ 2 marks)

(a) Verifiable:

A requirements document often becomes part of the contract between a client and a system developer. It describes what the developer will deliver and be paid for. Requirements should be verified against requirements, i.e. to check if the software meets the requirements.

(b) Complete

The requirements should be complete, and not leave out any areas. Current trends are to break the development up into smaller modules and implement each of them.

(c) Unambiguous

A requirement is ambiguous if it has more than one possible meaning. May be ambiguous due to poor choice of words and/or differing definitions of a word or phrase.

(d) Consistent

All parts of the requirements document should be consistent with each other. They should not contradict each other.

(e) Modifiable

The requirements should be structured so that it is possible to change it at little to no cost. Requires it to be structured carefully, e.g. separating functional from non functional requirements, supplying a glossary (a table of definitions)

(f) Traceable

The requirements should be structured so that it is possible to uniquely identify each requirement

- Each one should have a unique number
- This allows it to be referred to in final testing of the delivered software, and in discussions with a client.

ii.Explain the following terms as applied to object-oriented analysis

a) Requirement (2 marks)

Requirement is a description or statement of a function, feature or condition that a user seeks to have implemented in a system.

(b)Requirement specification (2 marks)

The requirements document is called the Software Requirements Specification (SRS). The SRS is a specification for a particular software product, program, or set of programs that perform certain functions in a specific environment iii.Six information to be included in a requirement specification (3 marks)

(a) An overview of the project

(b) Goals/purpose

(c) Glossary – definition all relevant terms

(d) System functions

- System functional requirements, these are functional requirements
- System attributes (non-functional requirements)

(e) Use cases – narrative descriptions of the domain processes

(f) Conceptual model – a model of important concepts and their relationships in the application domain.

iv. Differentiate the following terms:

(a) Functional requirements and non-functional requirements (4 marks)

- Functional Requirements
- Are those that relate directly to the functioning of the system.
- These are the aspects of the system the client is most likely to recognize.
- Non-functional requirements:
- They are constraints/restrictions imposed on the system

(b) Evident functions and hidden functions (3 marks)

- Evident functions are requirement to be performed and recognized by the user.
- Hidden functions are requirement to be performed by the system, but not visible to users.

v. With an example explain the concept of use case granularity (2 marks)

Use case granularity Is to keep use cases at a fairly "high level", by using the following rule-of-thumb in mind. "A Use Case should satisfy a goal for the actor"

vi. Two ways applied to identify use cases

(c) Use case identification on actor-based: method (2 marks))

- Find and specify all the actors by looking at which users will use the system and which other systems must interact with it.
- For each actor, identifying the processes they initiate or participate in by looking at how the actor communicate/interact with (or use) the system to do his work.

(d) Use case identification on event-based method (2 marks)

(a) Identify the external events that a system must respond to.

(b) Relate the events to actors and use cases.

vii. List six reasons why use cases are important (Any six) (3 marks))

(a) They answer what the system should do for each user or actor question.

(b) Use Cases define the scope of the system. They enable to visualize size and scope of the

entire development.

(c) They facilitate communication means between the customer and developers (use case diagram is so simple, anyone can understand it)

(d) They represent system functions and the relationship among the system functions in a narrative way.

(e) They extent the identification of concepts and objects involved in the application domain.

(f) Use Cases are very similar to requirements, but whilst requirement tend to be vague, confusing, ambiguous and poorly written, the tighter structure of Use Case tend to make them far more focused

(g) The "sum" of the use cases is the whole system. Anything not covered by a use case is outside the boundary of the system to be developed. So the Use Case diagram is complete with no holes

(h) Use Cases guide the development teams through the development process. They give the

summary of what is needed to be done by the system

(i) Use Cases provide a method for planning development work, and allow to estimate how long the development will take

(j) Use Cases provide the basis for creating system tests

(k) Use cases help with the creation of user guides

(l) Use cases can be a source of information during validation and verification of a software

viii. Three possible state change when a system operation is executed (1.5 marks)

(a) Old objects may be removed,

(b) New objects and links may be created, and

(c) Values for attributes of objects may be modified.

ix. Importance of a contract on object-oriented analysis (3 marks)

Contracts are used to describe detailed system behavior in terms of state changes to objects in the Domain Model, after a system operation has executed. We need to know how to write a contract for a system operation, the contract of an operation is defined mainly in terms of its pre-conditions and post-conditions.

x. How do "system input event" relate to "system operation" (3 marks)

When an actor interacts with the system for a certain use case, events are being generated to a system.

- A system input event is an external input generated by an actor to a system. A system input event initiates a responding operation
- A system operation is an operation that the system executes in response to a system input event.

Unit Readings and Other Resources

- Ariadne Training (2001), "UML Applied Object Oriented Analysis and Design Using the UML", Ariadne Training Limited
- Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1
- Larman C. (2004), "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", (3rd Edition) 3rd Edition, Prentice Hall; 3 edition (October 30, 2004), ISBN-13: 978-0131489066
- Liu Z., (2001), "Object-Otiented Software Development Using UML", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.
- Ojo A. and Estevez El., (2005), "Object-Oriented Analysis and Design with UML", Training Course, The United Nations University, UNU-IIST International Institute for Software Technology, e-Macao Report 19, Version 1.0, October.
- Pressman Roger S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783

Unit IV: Object-Oriented Design

Unit Introduction

After identifying requirements and creating a domain model, then methods are added to the software classes, and messaging between the objects to fulfill the requirements are defined. Deciding what methods belong where, and how the objects should interact, is terribly important. This is the heart of what it means to develop an object-oriented system, apart from drawing domain model diagrams.

Unit Objectives

Upon completion of this unit you should be able to:

1. Explain notion of collaboration diagrams

2. Apply UML notation for collaboration diagram

3. Describe the nature of the design phase

4. Use Patterns for assigning responsibilities to objects

5. Use Patterns to create collaboration diagrams

KEY TERMS

GRASP Pattern

The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in methodical, rational, explainable way. They describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience. This approach to understanding and using design principles is based on patterns of assigning responsibilities.

Software Design

Software design is an activity in which software components and their relationships are identified, based on a customer's requirements. Design is about how to solve a problem, so there is always a design process.

Software Design Process

The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models, and document the component interfaces.

Activity 1 – Interactive Diagrams – Collaboration Diagram

Introduction

A contract for system operations describes what the system operation does, but it does not show a solution of how software objects are going to work collectively to fulfill the contract of the operation. The later (how) is specified by an interaction diagrams in UML. A major task of the design phase is to create the interaction diagrams for the system operations.

Types of Interactive Diagrams

As explained in activity 2.2: UML diagrams, the UML define two kinds of interaction diagrams: Collaboration diagrams and sequence diagrams, either of which can be used to express similar or identical messages interactions.

Each type has strengths and weaknesses as shown as indicated in Table 4.1:

- When drawing diagrams to be published on pages of narrow width, collaboration diagrams have the advantage of allowing vertical expansion for new objects
- Additional objects in a sequence diagram must extend to the right, which is limiting.
- On the other hand, collaboration diagram examples make it harder to easily see the sequence of messages.

Table 4.1: Sequence Diagram Versus Collaboration Diagram

Type	Strengths	Weaknesses
Sequence Diagram	Clearly shows sequence or time ordering of messages	Forced to extend to the right when adding new objects; consumes horizontal space
	Simple notation	
Collaboration Diagram	Space economy – flexibility to add new objects in two dimensions	Difficult to see sequence of messages
	Better to illustrate complex branching, iteration, and concurrent behaviour	More complex notation

We have used system sequence diagrams in object oriented analysis, but now we are referring to another name object sequence diagram. Table 4.2 describes the difference between the system sequence diagram and object sequence diagram. You can note that the object sequence diagram is being used at design phase.

Table 4.2: Difference Between System Sequence Diagram and Object Sequence Diagram

System Sequence Diagrams	Object Sequence Diagrams
Illustrates the interaction between the whole system and external actors	Shows the interactions between objects of the system
Shows only system's external events and thus identifies system operations	Identifies operations of objects
Are created during the analysis phase	Are models created and used in the design phase

Collaboration Diagrams

A collaboration diagram is a graphical representation which shows the linkage between a number of objects and the links between them. Collaboration diagrams show the messages that are passed from one object to another. Since both collaboration diagrams and object sequence diagrams can express similar constructs, we can opt to use one over the other. We shall mainly discuss and use collaboration diagrams. As an example, collaboration diagram for "makePayment(cashTendered)" operation for the POST system is shown in Figure 4.1

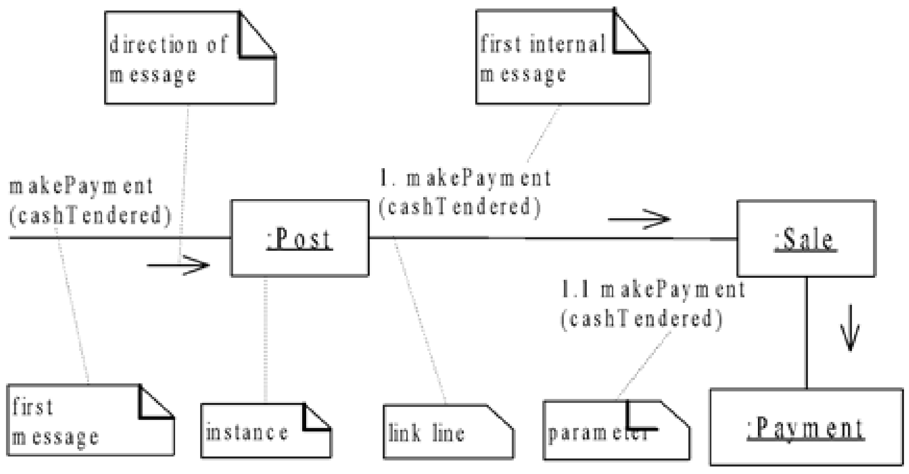


Figure 4.1: A Collaboration Diagram for makePayment(cashTendered) Operation

While in comparison with the object sequence diagram for "makePayment(cashTendered)" operation for the POST system we can have as shown in Figure 4.2.

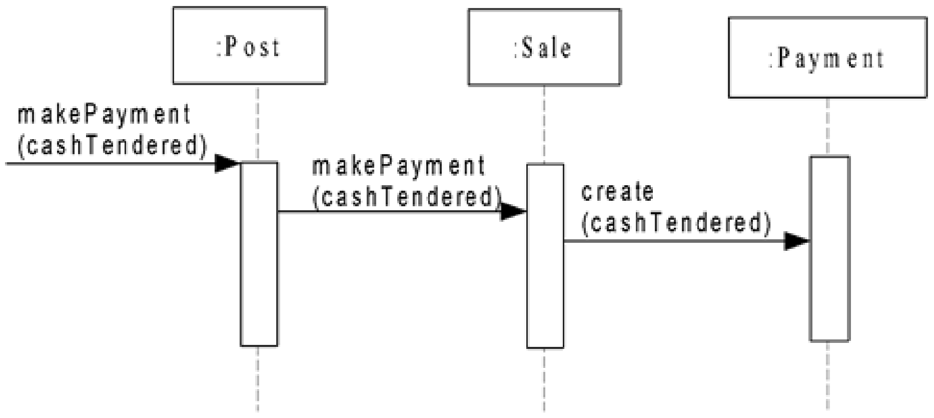


Figure 4.2: A Object Sequence Diagram for makePayment(cashTendered) Operation

Both of the two diagrams represent the following pseudo program (Liu, 2001)

- POST accepts a call of it method makePayment() (from an Interface Object);
- POST calls for the method makePayment() of :Sale;
- Sale calls the constructor of class Payment to create a new:Payment

UML Notation for Collaboration Diagrams

Collaboration diagrams have basic notations as presented in Figure 4.1. Basic notations include:

- Instances: is the same object but the name is being underlined and it has to always preceded by a colon.
- Links: this is a connection path between two objects to show some form of navigation and visibility instances.
- Message: messages are represented using an arrow on the link line. Messages are numbered to show the sequential order in which the message are sent.
- Parameter: Parameters are shown within parentheses following the message.

A collaboration diagram has more notations which can be used when need comes as follows (Liu, 2001):

Representing a return value

Some message sent to an object may require a returning message. A return value may be shown by preceding the message with a return value variable name and an assignment operator (':=') as shown in Figure 4.3. The standard syntax for messages is:

return := message(parameter : parameterType) : returnType

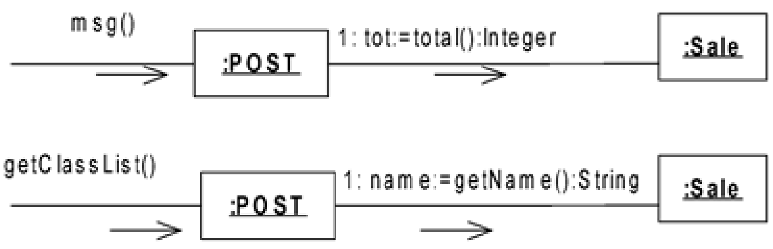


Figure 4.3: Return Values in Collaboration Diagram

Representing iteration

An object may repeatedly send a message to another object a number of times. This is indicated by prefixing the message with a start ('*') as in Figure 4.4.

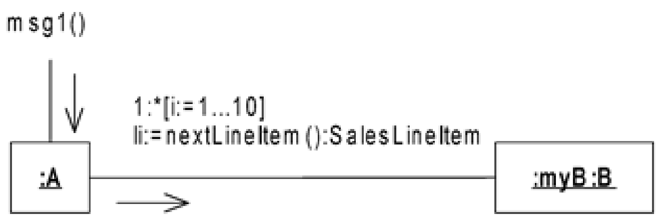


Figure 4.4: Representing Iteration in Collaboration Diagram**Representing creation of instances**

The UML creation message is create which is independent of programming languages, shown being sent to the instance being created. Optionally, the newly created instance may include a <<new>> symbol as shown in Figure 4.5. A create message can optionally take parameters when some attributes of the object to be created need to be set an initial value

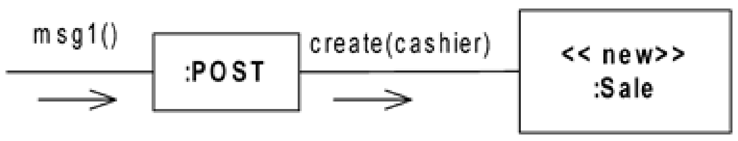


Figure 4.5: Creation of an Instance in Collaboration Diagram

Representing message number sequencing

The order of messages is illustrated with sequence numbers, as shown in Figure. The numbering scheme is:

i. The first message is not numbered. Thus, msg1() is unnumbered.

ii. The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have appended to them a number. Nesting is denoted by pre-pending the incoming message number to the outgoing message number.

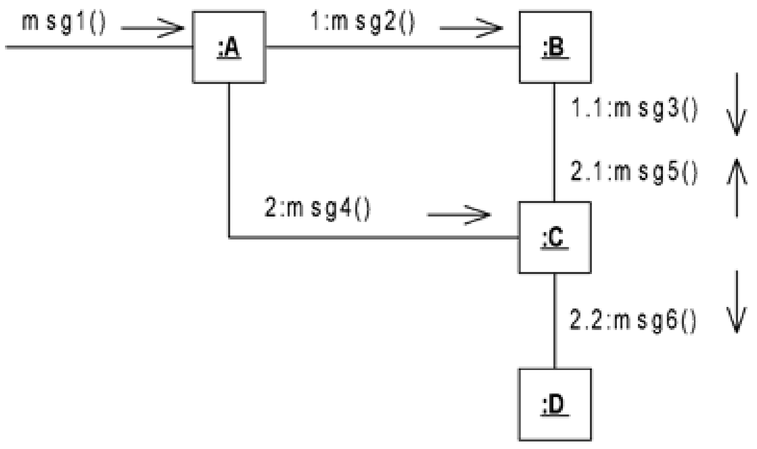


Figure 4.6: Message Number Sequencing in Collaboration Diagram

Representing conditional messages

Sometimes, a message may be guarded and can be sent from one object to another only when a condition holds. At a point during the execution of an object, a choice of several messages, guarded by different conditions, will be sent. In a sequential system, an object can send one message at a time and thus these conditions must be mutually exclusive. When we have mutually exclusive conditional messages, it is necessary to modify the sequence expressions with a conditional path letter.

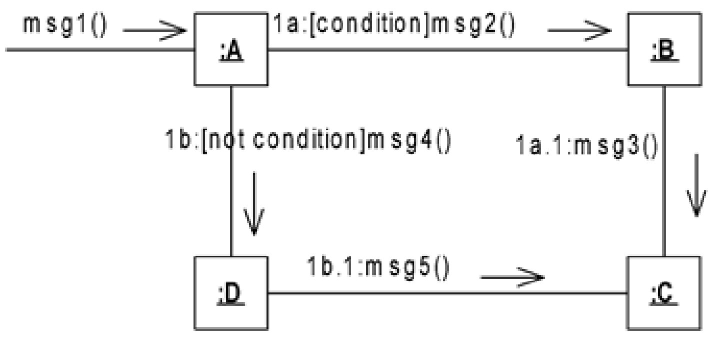


Figure 4.7: Conditional Message in Collaboration Diagram

Note that:

- Either 1a or 1b could execute after msg1() , depending on the conditions.
- The sequence number for both is 1, and a and b represent the two paths.
- This can be generalized to any number of mutually exclusive Conditional Messages.

Representing multiobjects

We call a logical set of instances/objects as a multiobject. A multiobject is an instance of a container class each instance of which is a set of instances of a given class (or type). E.g. SetOfSales is a class each instance of which is a set of sales. Each multiobject is usually implemented as a group of instances stored in a container or a collection object. In a collaboration diagram, a multiobject represents a set of objects at the "many" end of an association. In UML, a multiobject is represented as a stack icon as illustrated in Figure 4.8.

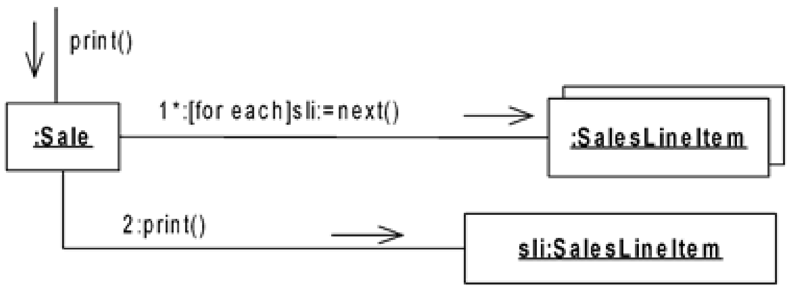


Figure 4.8: Multiobject and Message to Multiobject

Conclusion

Collaboration diagrams are mainly used in object-oriented design phase. They are flexible to add new concepts in two dimensions hence occupy less space. In design phase objects are supposed to be provided with operations to perform. Collaborations are easily used for this purpose with the help of a Pattern which facilitate assigning responsibility to objects.

Activity Assessment

i. Clearly describe basic UML notation used in collaboration diagrams.

ii. What is the difference between *msg(), *[true] msg(), and [true] msg() in a collaboration diagram?

iii. Develop a collaboration diagram in which:

(a) An object O receives message meg from an actor;

(b) O create a new object P;

(c) P sends a message to Q;

(d) then O destroy P

iv. Distinguish between the following terms

(e) Sequence diagram and collaboration diagram

(f) System Sequence diagram and object sequence diagram

Activity 1 – Overview of Design Phase

Introduction

Complex system operations must be decomposed into simpler internal operations which are assigned to (or carried out by) objects. The major task in the design is to create the collaboration diagrams for the system operations identified in the requirement analysis phase. The most important and difficult part in the generation of collaboration diagrams is the assignment of responsibilities to objects. This activity discusses the general principles for responsibility assignment, which are structured in a format called patterns.

Artifacts needed for creating collaboration diagrams:

(a) Conceptual model: In this, software classes corresponding to concepts are defined.

(b) System operations contracts: which deals with the identification of responsibilities and post conditions that the interaction diagrams must fulfill

(c) Essential (or real) use cases: Involves collection of information about what tasks the interaction diagrams fulfill, in addition to what is in the contracts

GRASP: Patterns for Assigning Responsibilities

Deciding what methods belong where and how objects should interact is terribly important and trivial. Assigning responsibility is a crucial/critical step in developing object-oriented systems.

By definition; a responsibility is a contract or obligation of an object. Responsibilities are related to the obligations of objects in terms of their behaviour. The purpose is to help methodically apply fundamental principles for assigning responsibilities to objects. Within the UML artifacts, a common context where these responsibilities (implemented as methods) are considered is during the creation of interaction diagrams. Responsibilities can be grouped into two main types (Liu, 2001):

Doing responsibilities:

These are about the actions that an object can perform including: doing something itself such as creating an object or doing a calculation, initiating an action or operation in other objects, controlling and coordinating activities in other objects

Knowing responsibilities:

These are about the knowledge an object maintains: knowing about private encapsulated data, knowing about related objects, knowing about things it can derive or calculate.

Creation of Collaboration Diagrams

When creating collaboration diagrams:

- Start with the responsibilities which are identified from the use cases, conceptual model, and system operations' contracts.
- Assign these responsibilities to objects, then:
- Decide what the objects need to do to fulfill these responsibilities in order to identify further responsibilities which are again assigned to objects.
- Repeat these steps until the identified responsibilities are fulfilled and a collaboration diagram is completed.

Responsibilities are assigned to classes of objects during object oriented design stage. Note that a responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities. Responsibilities of an object are implemented by using methods of the object which either act alone or collaborate with other methods and objects. For example, with reference to POST system (Liu, 2001):

- The Sale class might define a method that is performing printing of a Sale instance; say a method named print.,
- To fulfill that responsibility, the Sale instance may have to collaborate with other objects, such as sending a message to SalesLineItem objects asking them to print themselves.

Within UML, responsibilities are assigned to objects when creation a collaboration diagram and the collaboration diagram represents both of the assignment of responsibilities to objects and the collaboration between objects for their fulfillment. Example, as presented in Figure 4.9:

- Sale objects have been given a responsibility to print themselves, which is invoked with a print message.
- The fulfillment of this responsibility requires collaboration with SalesLineItem objects asking them to print.

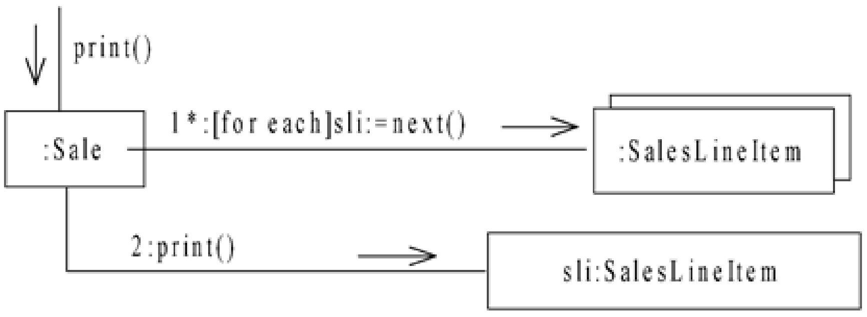


Figure 4.9: Performing Print Operation on Each Object of a Multiobject

General Principles in Assigning Responsibilities

Patterns are best principles for assigning responsibilities to objects. Patterns are guide to the creation of software methods. They are best principles for assigning responsibilities to objects. They are solutions to common occurring problems. In view of this module the GRASP patterns: General Responsibility Assignment Software Patterns is going to be applied. GRASP pattern takes the following format:

Pattern Name:	The name given to the pattern
Solution:	Description of the solution of the problem
Problem:	Description of the problem that the pattern solves

Most simply, a pattern is a named problem/solution pair that can be applied to new context, with advice on how to apply it in novel situations. GRASP patterns include the following five basic patterns (Ariadne, 2004 and Liu, 2001):

1. Expert,
2. Creator,
3. High Cohesion
4. Low Coupling and
5. Controller.

GRASP 1: Expert

This is a very simple pattern and is being used more than any other pattern in assignment of responsibilities. Expert class is that class that has information necessary to fulfill the responsible. Experts do things relating to the information they know. Assign a responsibility to the information expert — the class that has the information necessary to fulfill the responsibility. Example is as shown in Figure 4.10.:

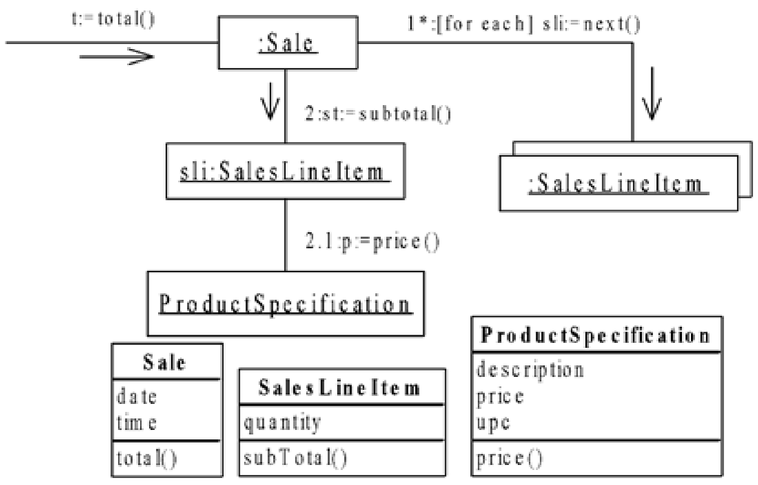


Figure 4.10: Collaboration Diagram for Getting the price of a product

Design Class	Responsibility
Sale	Knows sale total
SalesLineItem	Knows line item subtotal
ProductSpecification	Knows product prices

GRASP 2: Creator

The creation of objects is one of the most common activities in an OO system. It asks the question "who should be responsible for creating instances of a particular class?" Assign class B the responsibility to create an instance of class A if one of these is true:

- B contains A
- B records A
- B aggregates A
- B closely uses A
- B has the initializing data for A

For example in POST system, who should be responsible for creating a SalesLineIteM instance? By Creator pattern, we should look for a class that aggregates, contains SalesLineItem instances. Since Sale contains (aggregates) many SalesLine Item objects, the creator pattern suggests that Sale is the right candidate to have the responsibility of creating SalesLine Items instances as in Figure 4.11. This means "makeLineItem" method will be defined in Sale.

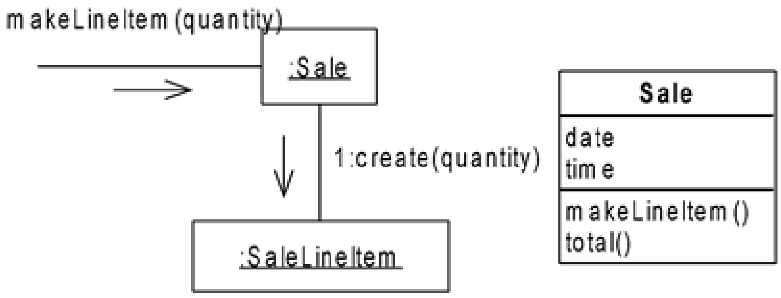


Figure 4.11: Collaboration Diagram for Create a SalesLineItem

GRASP 3: High Cohesion

Cohesion or coherence is the strength of dependencies within a subsystem. Cohesion is a measure of how strongly related and focused the responsibilities of a class. It is the internal "glue" with which a subsystem is constructed. A component is cohesive if all its elements are directed towards a task and the elements are essential for performing the same task. If a subsystem contains unrelated objects, coherence is low. High cohesion is desirable.

A class with high cohesion has highly related functional responsibilities, and does not do tremendous amount of work. Such classes have a small number of methods with simple but highly related functionality. In a good object-oriented design, each class should not do too much work. Assign few methods to a class and delegate parts of the work to fulfill the responsibility to other classes.

The following "liftController" class in Figure 4.12 is not well designed. A class does a lot of work.

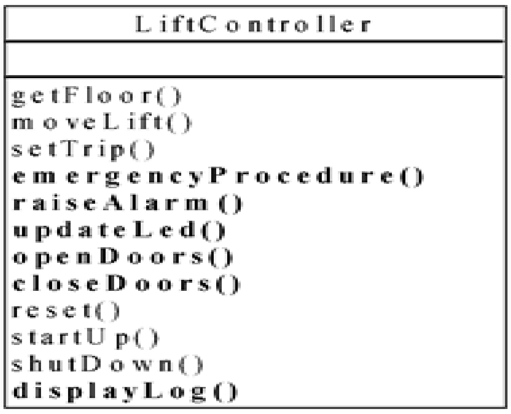


Figure 4.12: The Lift Controller Class with Low CohesionNote that a class should represent one "thing" from the real would. Key separate abstractions from liftController as shown in Figure 4.13 are:

- An Alarm
- Lift
- Doors and
- Logs

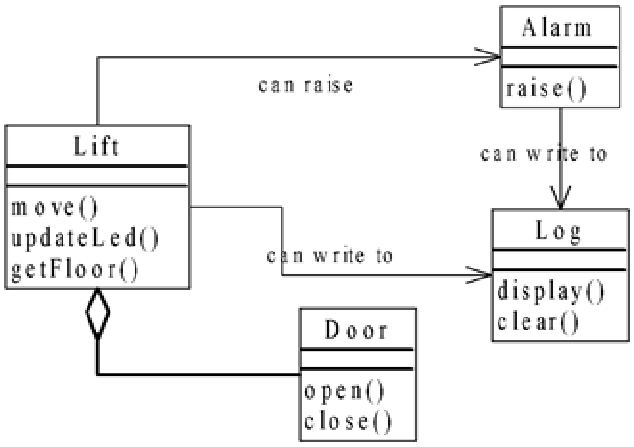


Figure 4.13: The Lift Controller Class Modelled as a Four Separate, More Cohesive Classes

GRASP 4: Low Coupling

Coupling is a measure of how strongly one class is connected to; has knowledge of; or relies on other classes. A class with low (or weak) coupling is not dependent on too many other classes. When we assign a responsibility to a class, we would like to assign responsibilities in a way so that coupling between classes remains low.

From the following example for "makePayment" use case and with the consideration of low coupling, the second design is preferable because it does not need an extra link formed between POST and Payment. This assignment of responsibility is also justifiable as it is reasonable to think that a Sale closely uses a Payment (Figure 4.14).

GRASP 5: Controller

Who handles a system event? Assign the responsibility for handling a system event message to a class representing one of these choices:

- Represents the overall system, device, or a subsystem (facade controller).
- Represents a use case scenario within which the system event occurs (use-case or session controller)

A controller is a non-user interface object responsible in handling a system input events, and the controller defines the method for the system operation corresponding to the system input event. One possible solution is to add/introduce a new class, and make it sit between the actor and the business classes. The name of this controller is usually called <name>Handler. Handler reads the commands from the user and then decides which classes the messages should be directed to. The handler is the only class that will be allowed to read and display. The system receive external input events, typically involving a GUI operated by a person.

Controller pattern takes the following form:

Table 4.3: Controller Pattern

Pattern Name	Controller
Solution	Assign the responsibility for handling a system (input) event to a class representing one of the following choices
	• Represents the "overall system" (facade controller).
	• Represents the overall business or organization (facade controller).
	• Represents something in the real-world that is active (for example, the role of a person) that might be involved in the task (role controller).
	• Represents an artificial handler of all system (input) events of a use case, usually named "< UseCaseName> Handler" (use-case controller).
Problem	Who should be responsible for handling a system input event?

Conclusion

A pattern is a well used procedure in collaboration diagrams. It is a solution to a common occurring problem. Five basis patters under GRASP patterns have been described, these are Expert, Creator, High Cohesion, Low Coupling and Controller. You need to be careful when using patterns so as to develop a modifiable and robust object-oriented design.

Activity Assessment

i. Explain clearly the following aspects of Object Model:

(a) Controller Pattern

(b) Cohesion Pattern

ii. A good piece of software should have high cohesion and low coupling. Explain.

Activity 2 – Application of Patterns in Design

Introduction

This activity shows how to apply GRASP patterns to assign responsibilities to object. It does the actual application in a POST case project system. As done by Liu (2001), the design is going to consider "Buy Item with Cash" and "Start Up" use cases.

From activity 4.2, the following can be highlighted as a guideline for making collaboration diagram: (Liu, 2001): The same guideline is summarized in Figure 4.15.

- Create a separate diagram for each system operation which has identified and whose contracts are defined.
- If the diagram gets complex, split it into smaller diagrams.
- Using the contract responsibilities and post-conditions, and use case description as a starting point, design a system of interacting objects to fulfill the tasks.
- Apply the GRASP to develop a good design.

Note that this areas if fully adopting the design made by Liu, (2001)

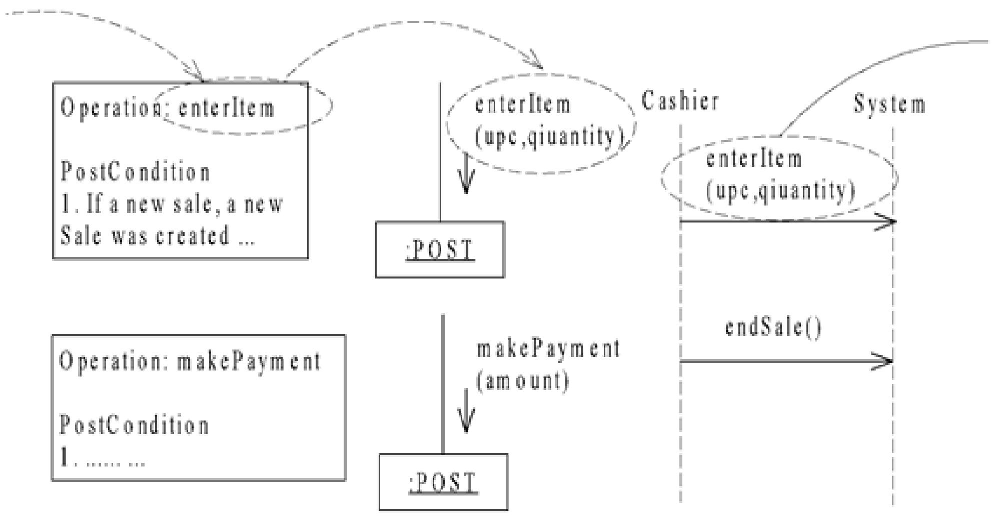


Figure 4.15: Guidelines for design

A Design of POST System

With "Buy Items with Cash" and "Start Up" use cases, we have identified four system operations enterItem, endSale, makePayment and startup as shown in Figure 3.13. According to our guidelines we should construct at least four collaboration diagrams. This activity will demonstrate only one operation and the remaining will be provided with the final collaboration diagrams. According to the Controller pattern, the POST class could be chosen as controller for handling these operations and hereby presented in Figure 4.16.

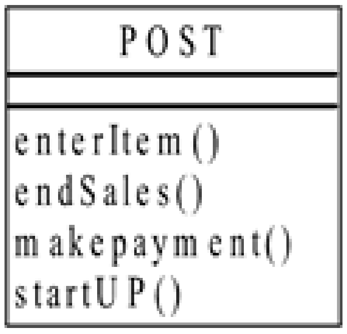


Figure 4.16: PPOST as a Controller Class

We need to look at the contract of enterItem what POST needs to do to fulfill this responsibility (See Table 4.4).

Table 4.4: Post-Condition for "enterItem" Operation

Post-conditions:	If a new sale, a Sale was created (instance creation).
	If a new Sale, the Sale was associated with the POST (association formed).
	A SalesLineItem was created (instance creation).
	The SalesLineItem.quantity was set to quantity (attribute modification).
	The SalesLineItem was associated with a ProductSpecification, based on UPC match (association formed).

Creating a New Sale:

- The post-conditions of enterItem indicate a responsibility for creation an object Sale.
- The Creator pattern suggests POST is a reasonable candidate creator of the Sale, as POST records the Sale.
- Having POST created the Sale; the POST can easily be associated with it over time.
- Creating a New SalesLineItem:
- When the Sale is created, it must create an empty collection to record all the future SaleLineItem that will be added.
- This collection will be maintained by the Sale instance, which implies by Creator that the Sale is a good candidate for creating it (SalesLineItem).

Figure 4.17 show the creation of a new sale by the POST controller class and a new SalesLine Item by a Sale class.

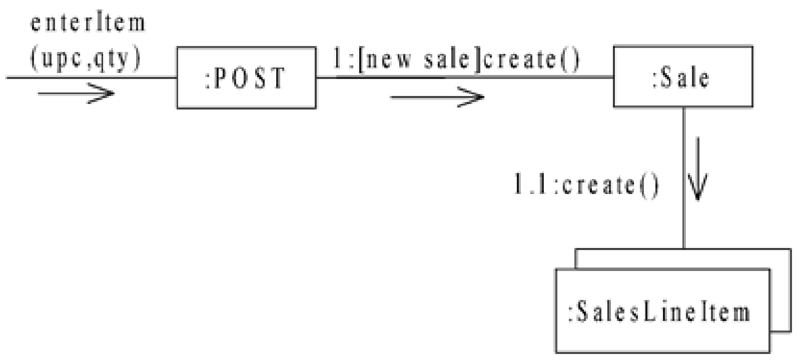


Figure 4.17: Collaboration Diagram to Create a New Sale and a New SalesLineItem

Finding a ProductSpecification

- Newly created SalesLineItem is required to be associated with ProductSpecification that matches with upc
- This need the parameters to the makeLineItem message sent to the Sale include ProductSpecification instance denoted by sec, which matches upc
- We need to retrieve the ProductSpecification before the message makeItem(spec,qty) is sent to Sale
- From Expert pattern, ProductCatalog contains all the productSpecification, hence good candidate for looking up the ProductSpecification

Visibility to a ProductCatalog

- From startUp() contract it shows POST was associated with the ProductCatalog
- POST is responsible in sending message to ProductCatalog denoted by specification

Hence the collaboration Diagram for enterItem will be as shown in Figure 4.18

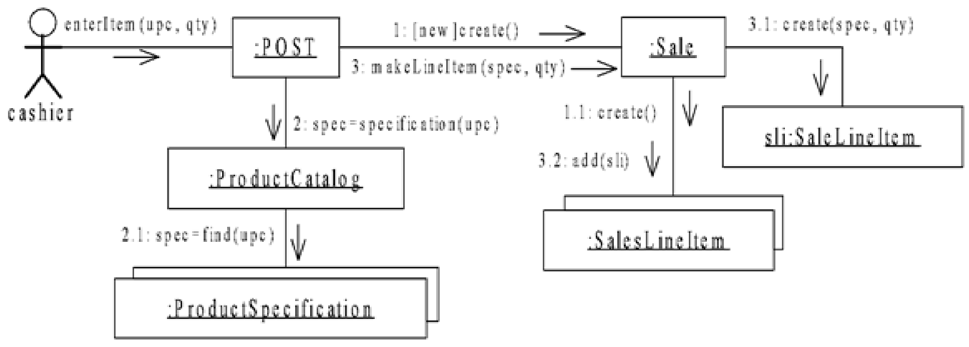


Figure 4.18: Collaboration Diagram for enterItem Operation

Collaboration Diagram for endSale will be as shown in Figure 4.19.

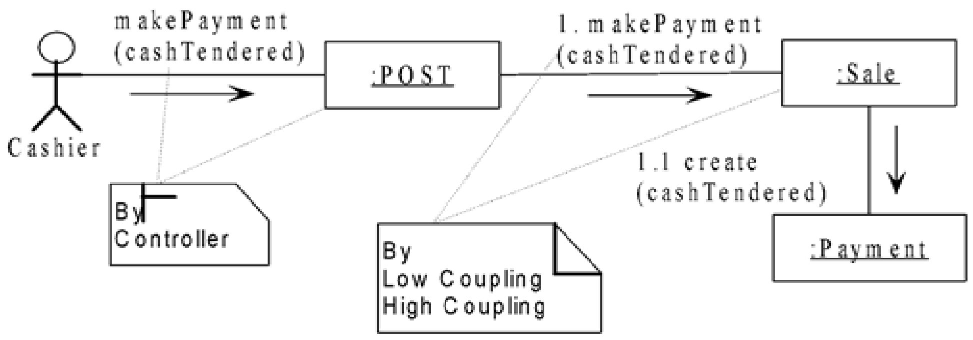

makePayment (cashTendered)
Cashier
:POST
1. makePayment (cashTendered)
:Sale
1.1 create (cashTendered)
:Payment
By Controller
By Low Coupling High Coupling

Figure 4.19: Collaboration Diagram for endSale Operation

Figure 4.19: Collaboration Diagram for endSale Operation

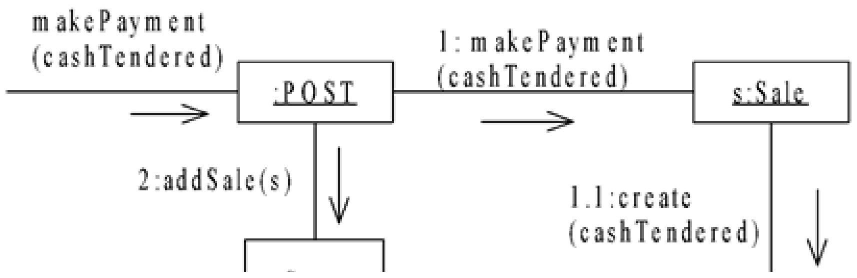


Figure 4.20: Collaboration Diagram for makePayment Operation

Collaboration Diagram for Logging a complete sale will be as shown in Figure 4.21

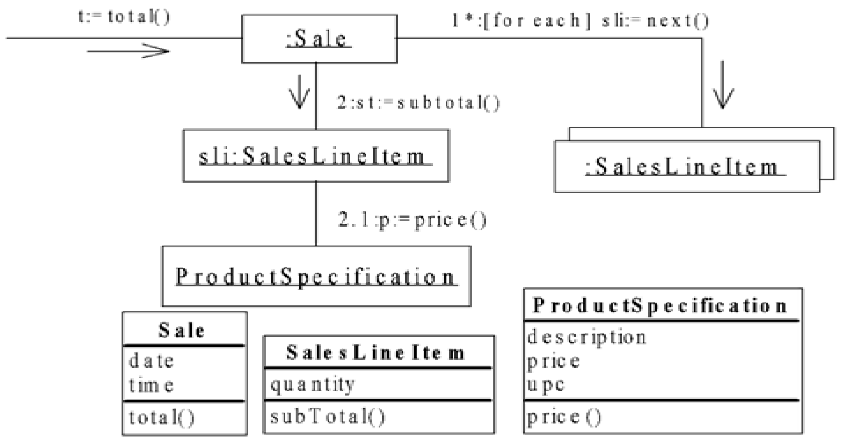


Figure 4.21: Logging a Complete Sale Collaboration Diagram

Collaboration Diagram for StartUp will be as shown in Figure 4.22.

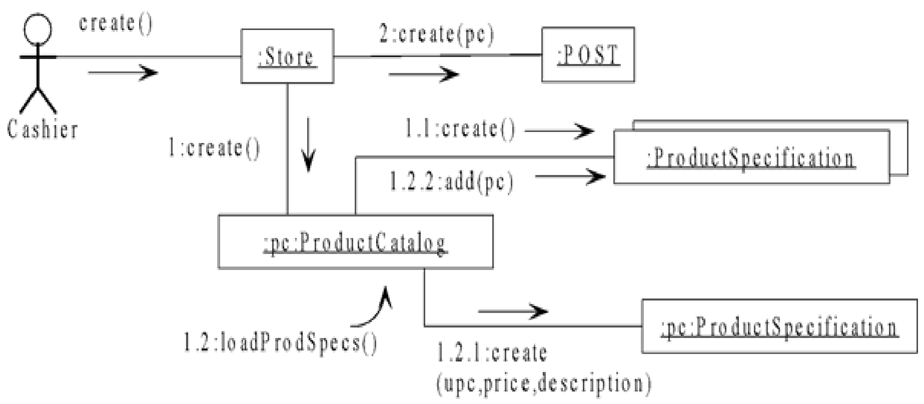


Figure 4.22: Collaboration Diagram for StartUp Operation

Conclusion

Application of a GRASP pattern has been demonstrated in this activity. The main input to this activity is the information from post condition of the system operation. We creating collaboration diagrams for a single system operation for the purpose of assigning responsibilities to collaborating object while performing a functionality.

Activity Assessment

Use the identified four system operation to do the following:

i. Bring forward the "post condition" obtained while creating system operation contracts

ii. Create collaboration diagram for the same four operations. With the help of GRASP assignment responsibility pattern identify methods for each class obtained

Activity 3 - Design Class Diagram

Introduction

During creation of a collaboration diagram, we record the methods corresponding to the responsibilities assigned to a class in the third section of the class box. These classes with methods are software classes representing the conceptual classes in the conceptual models. Then based on these identified software classes, the collaboration diagrams and the original conceptual model, we can create a design class diagram which illustrate the following information:

- Classes, associations and attributes
- Methods
- Attribute type information
- Navigability
- Dependencies

Note that the design part has been adopted from Liu, (2001)

Steps in Making a Design Class Diagram

Use the following strategy to create a design class diagram:

1. Identify all the classes participating in object interaction by analyzing the collaborations
2. Present them in a class diagram
3. Copy attributes from the associated concepts in the conceptual model
4. Add methods names by analyzing the interaction diagrams
5. Add type information to the attributes and methods
6. Add the association necessary to support the required attribute visibility
7. Add navigability arrow necessary to the associations to indicate the direction of the attribute visibility
8. Add dependency relationship lines to indicate non-attribute visibility
9. More:
    - Enhance attributes by adding datatypes
    - Determine visibility e.g. public (+), Private (–) – concept for encapsulation

Add Associations and Navigability

Each end of an association is called a role. In a design class diagram, the role may be decorated with a navigability arrow. Navigability is a property of the role which indicates that it is possible to navigate uni-directionally across the association from objects of the source to the target class as shown in Figure 4.23.

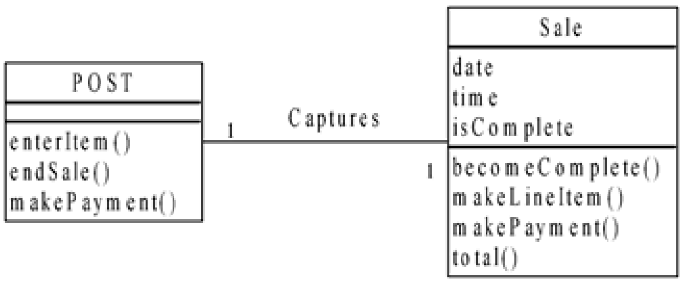


Figure 4.23: Part of Design Class Diagram Showing Navigability

Navigability is usually interpreted as attribute visibility from the source class to the target class. During the implementation in a OOP language it is usually translated as the source class having an attribute that refers to an instance of the target class. For example, the POST classes will define an attribute that reference a Sale instance.

DCD with Association and Navigability

Correcting all the activities done in previous units and activities, finally the design class Diagram will be as shown in Figure 4.24.

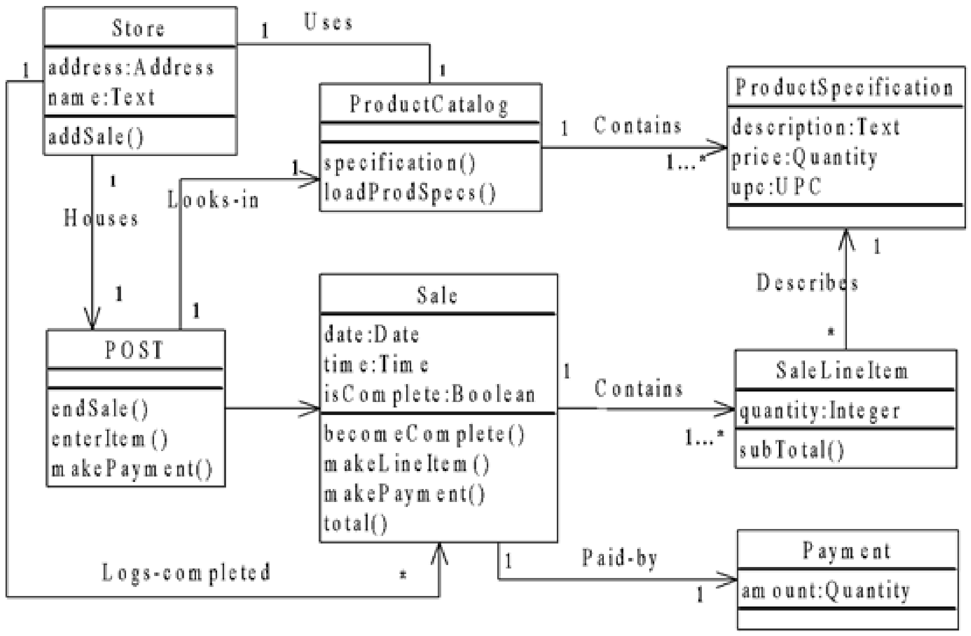


Figure 4.24: Design diagram with association and navigability

Conclusion

DCD is the final structure of the software designed. To make concepts coordinate, associations has to be considered and their multiplicity. Navigation into DCD is important to be shows as well.

Activity Assessment

i. Discuss the relationship between the conceptual model and the design class model in terms of their classes and associations. What are the possible ways to associate two classes in a design class diagram?

ii. From individual/group case studies, do the following:

(a) Create object sequence diagram for the four operations selected during object oriented analysis

(b) Create collaboration diagram for the same four operations used in question (a) above. With the help of GRASP assignment responsibility pattern identify methods for each class. (Compare with the related object sequence diagrams. note at this stage any of the two can be used as per the convenience)

(c) By using some of the information contained in the conceptual class diagrams, describe the attributes and operation of the software classes. Indicate the types for attributes and operations of the classes as well their visibilities

(d) Present your software classes in a design class diagram and indicate navigability on class associations

UNIT SUMMARY

In this unit five "GRASP" Pattern have been explored to show its importance in Object Oriented Design. Having a Design Class Diagram in place any implementation language can be applied to implement a DCD into code. As with structured approach to have the entity relation Diagram (ERD) to be the final structure for implementation, with object-oriented the final product is a Design Class Diagram ready for implementation using any means.

This module is opting to show how the DCD can be mapped with the typical object-oriented programming language. The language applied is Java, though any object oriented programming language can be applied, like C++.

Unit Assessment

Instructions

Answer the following questions.

i. Explain the difference between coupling and cohesion in application design, including explanation of why each is important. (Answer: Section 4.2.4).

ii. With reference to object oriented approach in system analysis and design, describe the followings:

(a) Two general types of responsibilities when using patterns (Answer: Section 4.2.2).

(b) The term "Responsibility" (Answer: Section 4.2.2).

(c) "Controller" patterns (Answer: Section 4.2.4).

iii. Distinguish between a system sequence diagram and an object sequence diagram (Answer: Section 4.1.2).

iv. Explain clearly why "Patterns" are needed while using Object Oriented Analysis and Design approach to software development? (Answer: Section 4.2.2).

Answer

i. The difference between coupling and cohesion in application design, including explanation of why each is important. (Each 2 marks)

(a) Coupling: Coupling is a measure of how strongly one class is connected to; has knowledge of; or relies on other classes.

Importance:

When we assign a responsibility to a class, we would like to assign responsibilities in a way so that coupling between classes remains low. That is to make a class is not to depend on too many other classes.

(b) High Cohesion: Cohesion is a measure of how strongly related and focused the responsibilities of a class. It is the internal "glue" with which a subsystem is constructed.

Importance:

When we assign a responsibility to a class, we would like to assign responsibilities in a way so that cohesion of a classes remains high. That is to make a class perform highly related functional responsibilities, and not do tremendous amount of work.

ii. With reference to object oriented approach in system analysis and design, describe the followings

(a) Two general types of responsibilities when using patterns (@ 2 marks)

Doing responsibilities:

These are about the actions that an object can perform including: doing something itself such as creating an object or doing a calculation, initiating an action or operation in other objects, controlling and coordinating activities in other objects

Knowing responsibilities:

These are about the knowledge an object maintains: knowing about private encapsulated data, knowing about related objects, knowing about things it can derive or calculate.

(b) The term "Responsibility" (2 marks)

A responsibility is a contract or obligation of an object. Responsibilities are related to the obligations of objects in terms of their behaviour.

(c) "Controller" patterns

A controller is a non-user interface object responsible in handling a system input events. The controller defines the method for the system operation corresponding to the system input event.

iii. Difference between a system sequence diagram and an object sequence diagram is (3 Marks) (Answer: Section 4.1.2)

System Sequence Diagrams	Object Sequence Diagrams
Illustrates the interaction between the whole system and external actors	Shows the interactions between objects of the system
Shows only system's external events and thus identifies system operations	Identifies operations of objects
Are created during the analysis phase	Are models created and used in the design phase

iv. “Patterns” are needed while using Object Oriented Analysis and Design approach to software development since (2 marks):

Patterns are guide to the creation of software methods. They are best principles for assigning responsibilities to objects. They are solutions to common occurring problems.

Unit Readings and Other Resources

- Larman C. (2004), “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”, (3rd Edition) 3rd Edition, Prentice Hall; 3 edition (October 30, 2004), ISBN-13: 978-0131489066
- Liu Z., (2001), “Object-Otiented Software Development Using UML”, The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.
- Ojo A. and Estevez El., (2005), “Object-Oriented Analysis and Design with UML”, Training Course, The United Nations University, UNU-IIST International Institute for Software Technology, e-Macao Report 19, Version 1.0, October.
- Pressman Roger S., (2001), “Software Engineering, A Practitioner’ S Approach” Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783

Unit V- Implementing Object-Oriented Designs into Programming Languages

Unit Introduction

The UML artifacts created during the design phase: the interaction diagrams and DCDs will be used as input to the code generation process. Implementation consists of artifacts such as the source code, database definitions, JSP/XML/HTML pages etc. Code created is part of the implementation model.

Java is used for the examples because of its widespread use and familiarity. However, this is not meant to imply a special endorsement of Java; any other object oriented programming language can be used like C++, Visual Basic, C#.

Implementation in an object-oriented programming language requires writing source code for:

- Class definitions – define the classes in the design class diagram in terms of the programming notation.
- Method definitions – define the methods of classes in the design class diagram in terms of the programming notation.

Unit Objectives

Upon completion of this unit you should be able to:

1. Describe the notion of the interface of a class and its features

2. Define a class in a programming language

3. Define a method of a class in a programming language

KEY TERMS

Software implementation

Software implementation in object-oriented system analysis and design approach is the creation of code. Implementation is the process of realizing the design as a program.

Reference attribute

A reference attribute is an attribute that refers to another complex object, not to a primitive type such as a String, Number and so on

Activity 1 - Notation for Class Interface Details

Introduction

The interface of a class primarily consists of the declarations of all the methods (or operations) applicable to instances of this class, but it may also include the declaration of other classes, constants, variables (attributes), and initial values. Therefore, the interface of a class provides its outside view and emphasizes the abstraction while hiding its structure and the secrets of its behaviour. By contrast, the implementation of a class is its inside view, which encompasses the secretes of its behaviour. The implementation of a class primarily consists of the definitions of all the methods (or the implementation of all the operations) declared in the interface of the class.

The Interface of a Class

The interface of a class can be in general divided into three parts

a. Public: A declaration that is accessible to all the clients which are the classes that have attribute visibility to this class.

b. Protected: A declaration that is accessible only to the class itself, its subclasses, and its friends

c. Private: A declaration that is accessible only to the class itself and its friends.

The UML provides a rich notation to describe features of the interface of a class. Attributes are assumed to be private by default. The notation for other kind of interface declarations is shown in Figure 5.1.

Similarly, Figure 5.2 shows some interface information about the classes in the POST system. Notice that except for loadProdSpecs() which is a private method of ProductCatalog, all other methods are private

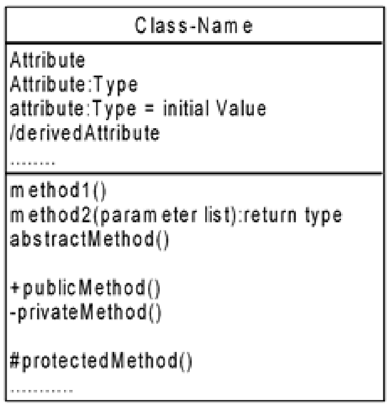


Figure 5.1: Notation for Class Interface Declaration

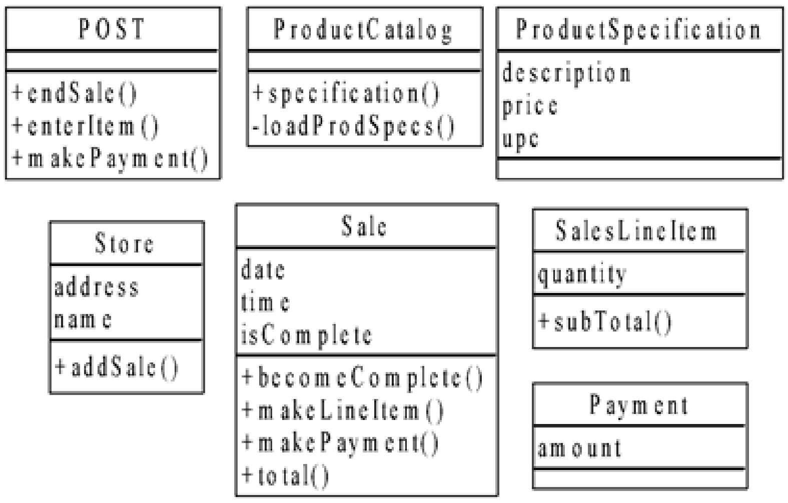


Figure 5.2: Interface Details for Post System Classes

Conclusion

Object-oriented system analysis and design follows all basis knowledge available in object oriented programming language like Java or C++. It is important to clearly show parts of class interfaces in terms of public, protected and private.

Activity Assessment

i. Differentiate between "public" and "private" terms as applied to object oriented

ii. With reference to classes in a design class diagram of your individual/group case projects, provide interface details

Activity 2 - Mapping a Design to Code

Introduction

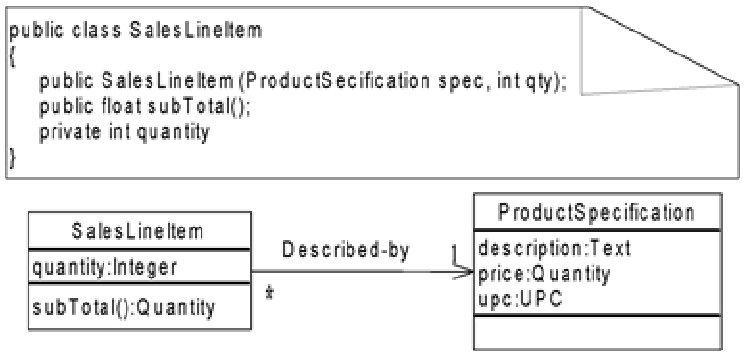
At the very least, design class diagrams depict the class name, superclasses, method signatures, and simple attributes of a class. This is sufficient to create a basic class definition in an object-oriented programming language. More information can be added later on after such a basic definition is obtained.

Defining a Class with Methods and Simple Attributes

Consider the class SalesLineItem and the partial design class diagram in Figure 5.3. A mapping of the class box for SalesLineItem in the design diagram to the basic attribute definitions and method signatures for the Java definition of SalesLineItem is straightforward.

Note that, we had to add the Java constructor SalesLineItem(….) because of the fact that a create(spec,qty) message is sent to a SalesLineItem in the enterItem collaboration diagram. This indicates, in Java, that a constructor supporting these parameters is required.

Observe also that the return type for the subtotal method was changed from Quantity to a simple float. This implies the assumption that in the initial coding work, the programmer does not want to take the time to implement a Quantity class, and will defer that.



```
public class SalesLineItem
{
    public SalesLineItem(ProductSecification spec, int qty);
    public float subTotal();
    private int quantity
}
```

Figure 5.3: SalesLineItem in Java

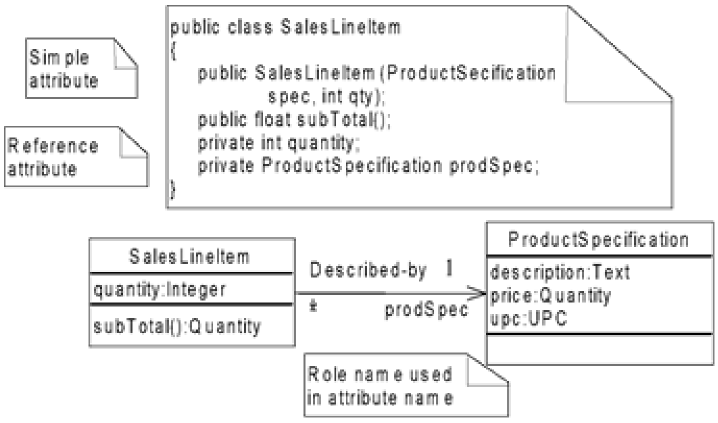
Add Reference Attributes

A reference attribute is an attribute that refers to another complex object, not to a primitive type such as a String, Number and so on.

The reference attribute of a class are suggested by the associations and navigability in a design class diagram.

For example, a SalesLineItem has an association to a ProductSpecification, and with navigability to it. This navigability is needed for sending the message price to the ProductSpecification from the SalesLineItem in the collaboration diagram for the total of the Sale. In Java, this means that an instance variable referring to a productSpecification instance is suggested.

Reference attributes are often implied, rather than explicit, in a class diagram. Sometimes, if a role name for an association is present in a class diagram, we can use it as the basis for the name of the reference attribute during code generation. The Java definition of the class SalesLineItem with a reference attribute prodSpec is shown in Figure 5.4.



```
public class SalesLineItem
{
    public SalesLineItem(ProductSecification
            spec, int qty);
    public float subTotal();
    private int quantity;
    private ProductSpecification prodSpec;
}
```

Figure 5.4: Add Reference Attributes

In the same way as we have defined the Java class for SalesLineItem, we can define the Java class for POST. This is shown in Figure 5.6.

Defining a Method from a Collaboration Diagram

A collaboration diagram shows the messages that are sent in response to a method invocation. The sequence of these messages translates to a series of statements in a method definition.

For example, Recall the collaboration diagram for the enterItem operation given in Figure 5.5. We should declared enterItem as a method of the POST class: public void enterItem(int upc, int qty)

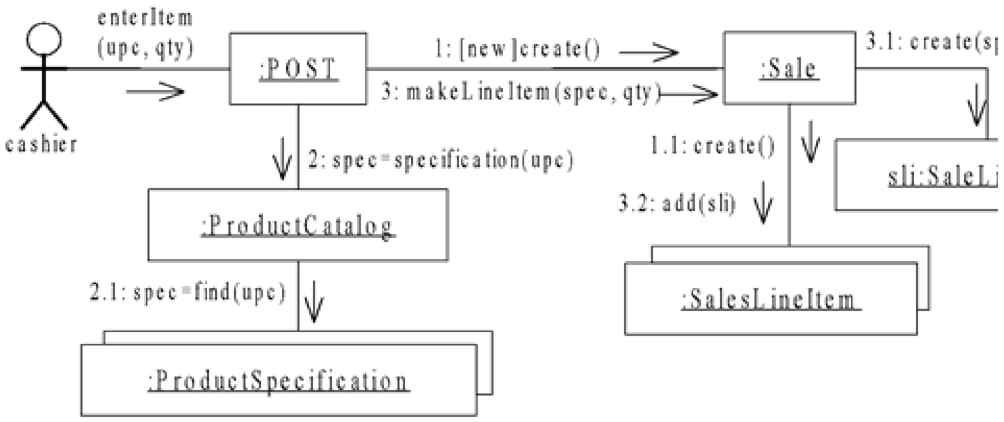
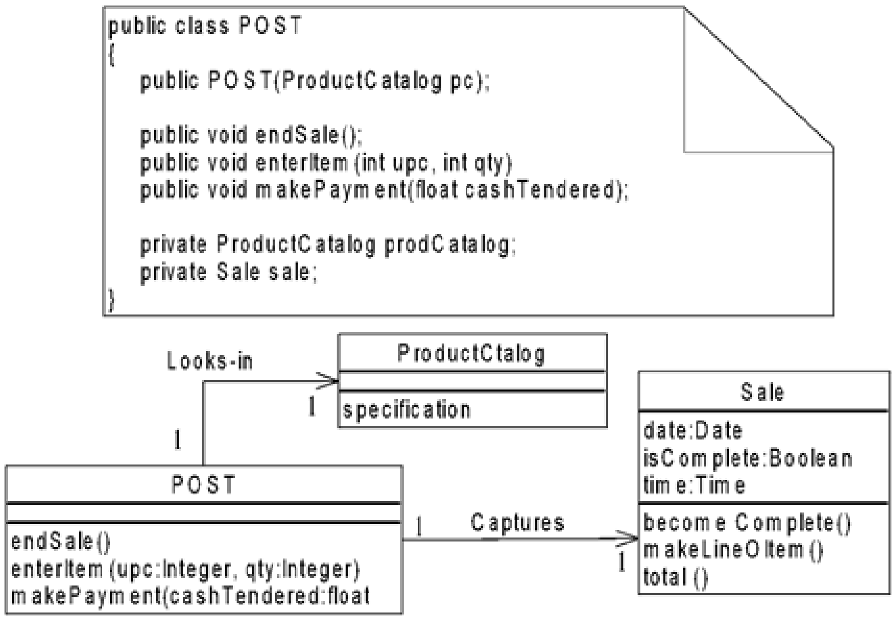


Figure 5.5: The enterItem Collaboration Diagram

Then we have to look at the messages sent by POST in response to the message enterItem received by POST as in Figure 5.6.



```
public class POST
{
    public POST(ProductCatalog pc);

    public void endSale();
    public void enterItem (int upc, int qty)
    public void makePayment(float cashTendered);

    private ProductCatalog prodCatalog;
    private Sale sale;
}
```

Figure 5.6: The Post Class in Java

Message 1: According to the collaboration diagram. In response to the enterItem message, the first statement is to conditionally create a new Sale

```
if (isNewSale()) {sale = new Sale() ; )
```

This indicates that the definition of the POST – enterItem method, needs the introduction of a new method to the POST class: isNewSale. This is a small example of how, during the coding phase, changes from the design will appear. It is possible that this method could have been discovered during the earlier solution phase, but the point is that changes will inevitably arise while programming.

As a first attempt, this (private) method will perform a test based on whether or not the sale instance variable is null (i.e. sale does not point to anything yet).

```
private Boolean isNewSale()
{
    return ( sale == null);
}
```

You may wonder why not simply hard-core this test into the enterItem method? The reason is that it relies on a design decision about the representation of information. In general, expressions that are dependent on representation decisions are best wrapped in methods so that if the representation changes, the change impact is minimized, Furthermore, to a reader of the code, the isNewSale test is more informative in terms of semantic intent that the expression

```
if (sale == null)
```

On reflection, you will realize that this test is not adequate in the general case. For example, what if one sale has completed, and a second sale is about to begin. In that case, the sale attribute will not be null; it will point to the last sale. Consequently, some additional test is required to determine if it is a new sale. To solve this problem, assume that if the current sale is in the complete state, then a new sale can begin. If it turns out later this is an inappropriate business rule, a change can be easily made. Therefore,

```
private Boolean isNewSale()
{
    return ( sale == null ) || ( sale.isComplete() );
}
```

Based on the above coding decisions, the new method isNewSale needs to be added to the POST class definition given in Figure 5.5. The design class diagram depicting the POST class should be updated to reflect this code change.

Message 2: The second message sent by the POST is specification to the ProductCatalog to retrieve a ProductSpecification:

ProductSpecification spec = prodCatalog.specification(upc);

Notice the use of the reference attribute prodCatalog.

Message 3: The third message sent by POST is the makeLineItem to the Sale:

sale.makeLineItem(spec, qty);

In summary, each sequenced message within a method, as shown on the collaboration diagram, is mapped to a statement in the Java method. Therefore, the complete enterItem of POST method is given as:

```
public void enterItem(int upc, int qty)
{
    if (isNewSale()) { Sale = new Sale(); }
    ProductSpecification spec = prodCatalog.specification(upc);
    sale.makeLineItem(spec, qty);
}
```

Container/Collection Classes in Code

It is often necessary for an object to maintain visibility to a group of other objects; the need for this is usually evident from the multiplicity value in a class diagram – it may be greater than one. For example, a Sale must maintain visibility to a group of SalesLineItem instances as shown in Figure 5.7.

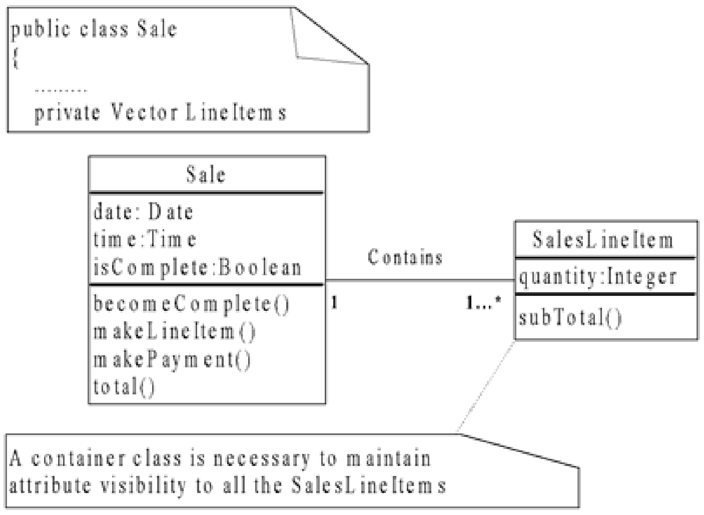


Figure 5.7: Reference to a Container

In object-oriented programming languages, these relationships are often implemented with the introduction of an intermediate container or collection. The one-side class in the association defines a reference attribute pointing a container/collection instance, which contains instances of the many-side class.

The choice of a container class is influenced by the requirements; key-based lookup requires the use of a Hashtable, a growing ordered list requires a Vector, and so on.

With these discussions, we can now define the makeLineItem method of the Sale class as shown in Figure 5.7.

Note that there is another example of code deviating from the collaboration diagram in this method: the generic add message has been translated into the Java-specific addElement message.

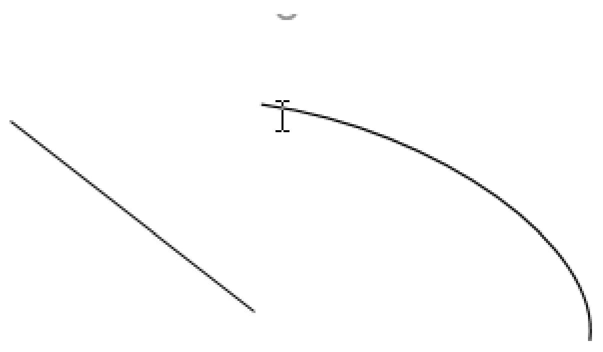


Figure 5.7: Sale – makeLineItem Method

Exceptions and Error Handling

Error handling has been ignored so far in the development of a solution, as we would like to focus on the basic questions of responsibility assignment and object-oriented design. However, in real application development, it is wise to consider error handling during the design phase. For example, the contracts can be annotated with a brief discussion of typical error situations and the general plan of response.

The UML does not have a special notation to illustrate exceptions. Rather, the message notation of collaboration diagrams is used to illustrate exceptions. A collaboration diagram may start with a message representing an exception handling.

Order of implementation

Classes need to be implemented (and ideally, fully unit tested) from least coupled and most coupled. For example, in the POST system, possible first classes to implement are either Payment or ProductSpecification. Next are classes only dependent on the prior implementations ProductCatalog or SalesLineItem.

Conclusion

The implementation made use of an object oriented programming language, however any means can be used implementation as long as a the final DCD is present.

Activity Assessment

i. Define a Java class for ProductCatalog

ii. Define a Java class for Store.

iii. Define a Java class for Sale

iv. From individual/group case studies, create a Java code from the developed case project DCD. (It can be part of it as well)

UNIT SUMMARY

Having a DCD in palce, any means of implementation can be adopted. This due to the fact that for many years, the term object oriented (OO) was used to denote a software development approach that used one of a number of object-oriented programming languages (e.g., Java, C++). Today, the OO paradigm encompasses a complete view of software engineering. Meaning that the OO is emphasizing more on the process which will lead to a quality software if designed following the software development process.

Unit Assessment

Instructions

Answer the following questions:

i. List and Describe three parts of the class interface (Answer: section 5.1.2)

ii. What does the term "protected" means in a class (Answer: section 5.1.2)

iii. What the term "reference attribute" mean (Answer: section 5.2.3)

Answers

i. Three parts of the class interface are(@ 2 marks)

(a) Public: A declaration that is accessible to all the clients which are the classes that have attribute visibility to this class.

(b) Protected: A declaration that is accessible only to the class itself, its subclasses, and its friends

(c) Private: A declaration that is accessible only to the class itself and its friends.

ii. The term "protected" in a class means a declaration that can be accessible only to the class

itself, its subclasses, and its friends (2 marks)

iii. The term "reference attribute" mean (2 marks)

- An attribute that refers to another complex object, not to a primitive type such as a String, Number and so on.
- The reference attribute of a class are suggested by the associations and navigability in a design class diagram.

Unit Readings and Other Resources

- Agarwal B. B., Tayal S. P. and Gupta M., (2010), "Software Engineering & Testing, an Introduction", Jones and Bartlett Publishers, ISBN: 978-1-934015-55-1
- Larman C. (2004), "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", (3rd Edition) 3rd Edition, Prentice Hall; 3 edition (October 30, 2004), ISBN-13: 978-0131489066
- Liu Z., (2001), "Object-Otiented Software Development Using UML", The United Nations University, UNU-IIST International Institute for Software Technology, Tech Report 229.
- Ojo A. and Estevez El., (2005), "Object-Oriented Analysis and Design with UML", Training Course, The United Nations University, UNU-IIST International Institute for Software Technology, e-Macao Report 19, Version 1.0, October.
- Pressman Roger S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783

COURSE SUMMARY

Object-oriented system analysis and design is the model driven approach and the modeling is being done by the help of the UML as a notation language. Several view points as presented by the UML has been discussed. We can summarize the ccategories of UML Documents with respect to SDLC from the first phase of requirement capture through implementation as shown in Table C.1. If the approach will be systematically followed the design will be well produced.

Table C.1: Categories of UML Documents for SDLC

Development Phase	Static	Behavioral
Requirements		Use Cases
		High Level Use Cases
		Use Case Diagram
Analysis	Domain or Conceptual or Class Diagram	
		System Sequence Diagrams
		Operation Contracts
Design		Collaboration Diagrams or Object Sequence Diagram
	Design Class Diagrams	

Course Assessment

i. Briefly explain the following terms in object-oriented software engineering context:

(a) use case,

(b) aggregation,

(c) association,

(d) specialization,

(e) abstraction.

ii.Explain what categories of information are included in "post-condition" field of the operation contract.

iii. What are the differences and relationships between Functional Requirements and use cases? How can they be identified?

iv. Briefly describe five GRASP Patterns. Whenever possible provide an example

v. Object-oriented development methods are rapidly replacing older structured development methods. Has structured development failed and why should object oriented development prove to be any better?

vi. Design a class called PhoneNumber, which represents a phone number then implement it in C++. A phone number has three parts country code, city code and the line number. For example given the phone number 255-22-2708624, the country code is 255, the city code is 22 and the line number is 2708624. There should be a way to access all the three parts of a phone number. Once a phone number object is created, the city code and the line number cannot be changed. However the country code may be changed. The phone number class should have a method called toString (), which returns a string in this format (country code)-city code-line number (i.e. left bracket, country code, right bracket, dash, city code, dash, line number)

vii. Total (T) Ltd wishes to computerise their petrol pumps. The program to be designed must start in wait state. On pressing any key there shall be a request to put the pump nozzle into the petrol tank. There are FOUR types of fuel available: LPG at a cost of 800 Tshs per litre. Unleaded at a cost of 700 Tshs per litre. Diesel at a cost of 850 Tshs per litre, Premium Unleaded at a cost of 900 Tshs per litre. Your program should cause the user to select the type of fuel required. The user must be able to :

i. Cancel the transaction prior to and during money input.

ii. Enter the amount of fuel required. For safety reasons only a maximum of 50 litres can be delivered at a time.

iii.Prepay for the fuel

After dispensing there must be a thank you message and an indication of any overpaid money that will be returned. There must be a request to return the nozzle to its holder. The program must return to its wait state.

(a) Identify at least 10 classes for the program

(b) Identify at least four use cases then draw a use-case model

viii. A program is required to run the controller of a burglar alarm system. A typical system consists of a number of sensors connected by individual circuits to a central control box containing the controller. The control box has a simple keypad and display. Sensors include switches, heat detectors and motion detectors. Each sensor has an identification code which can be read by the controller to identify the sensor.

The controller allows an operator to select which sensors are active and turn on or off the system. If a sensor is triggered when the system is active, the controller must activate the alarms (a siren and a bell) and display a message on the display panel indicating which sensor is involved. The operator must enter a security code before the system is turned on or off."

(a) Create a detailed UML class diagram framework for the alarm program.

(b) Draw a use case model consisting of 4 use cases and actors that interact with them.

Describe each use case in 1-3 sentences.

ix. Consider the following scenario

STREPT is an academic system. Some of the exercises concern a student record system. Students belong to departments like Computing or Electronics. Departments are organized into faculties, like Science or Engineering. Students study for a qualification like a Degree in Mathematics or a Post-Graduate Diploma in Holistic Synergizing. To obtain a qualification, students take courses.

Courses last for an academic year and consist of one or more modules like NumberTheory or PropositionalCalculus. Each module is worth a number of credit points and each course has a required credit points total. Courses and modules vary a little bit, year by year. Credit points, prerequisites and durations might be modified, for example. A presentation is what the actual delivery of course or module is called.

a) Write at least eight (8) functional requirements for STREPT academic system. For each functional requirement provide the requirement cross references, description, and category.

(b) Apart from "System Administrator" role, identify at least two more possible actors of this system, and describe their role to play

(c) Identify major use cases for this system and Draw a simple use case diagram showing identified possible actors and use cases for each

x.Pretend that you are going to build a new system that automates or improves the interview process for the career services department of your school.

(a) Develop a requirements definition for the new system. Include both functional and only two (2) important nonfunctional system requirements.

(b) Describe two possible main actors

(c) Identify use cases and represent it using a use case diagram for two identified actors in 5 (ii)

(d) From a use case diagram presented above, provide a detailed use case description for any two of the use cases related to the basic functioning of the interview process using the use case description template.

Course References

- Ariadne Training (2001), "UML Applied Object Oriented Analysis and Design Using the UML", Ariadne Training Limited
- Bjork R. C., (2004), "ATM Simulation", ATM Online", URL: http://www.math-cs.gordon.edu/courses/cps211/ATMExample/
- Booch G., Rumbaugh J. and Jacobson I. (2005), " Unified Modeling Language User Guide", Addison Wesley , Second Edition
- Larman C. (2004), "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development", (3rd Edition) 3rd Edition, Prentice Hall; 3 edition (October 30, 2004), ISBN-13: 978-0131489066
- Liu, Z. (2001), "Object-Oriented Software Development Using UML", March, The United University – International Institute for Software Technology (UNU/IIST), Report No. 229.
- Nellen, T. and Mayo, L. (2000), "We Learn by Doing", URL: http://english.ttu.edu/kairos/5.1/coverweb/nellenmayo/doing.html.
- Ojo A. and Estevez E., (2005), "Object-Oriented Analysis and Design with UML", Training Course, The United Nations University, UNU-IIST International Institute for Software Technology, e-Macao Report 19, Version 1.0, October.
- Pressman Roger S., (2001), "Software Engineering, A Practitioner' S Approach" Fifth Edition, McGraw-Hill Higher Education, ISBN 0073655783
- Sommerville Ian (2000), "Software Engineering (6th Edition)". Addison-Wesley, Boston USA

The African Virtual University Headquarters

Cape Office Park

Ring Road Kilimani

PO Box 25405-00603

Nairobi, Kenya

Tel: +254 20 25283333

contact@avu.org

oer@avu.org

The African Virtual University Regional Office in Dakar

Université Virtuelle Africaine

Bureau Régional de l'Afrique de l'Ouest

Sicap Liberté VI Extension

Villa No.8 VDN

B.P. 50609 Dakar, Sénégal

Tel: +221 338670324

bureauregional@avu.org